Escuela Politécnica Superior

Aplicación Android para la realización de tours guiados

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Martín Laiz Gómez

Tutor/es:
Antonio Javier Gallego Sánchez

Septiembre 2019

Universitat d'Alacant
Universidad de Alicante

Aplicación Android para la realización de tours guiados

Autor
Martín Laiz Gómez

Director
Antonio Javier Gallego Sánchez
Departamento de Lenguajes y Sistemas Informáticos

Grado en Ingeniería Informática

Escuela Politécnica Superior

Universidad de Alicante
ALICANTE, 3 de septiembre de 2019

Justificación y Objetivos

Teniendo en cuenta la gran importancia que tiene el turismo en España y sobre todo en la Costa Blanca, donde nos encontramos, he desarrollado una aplicación móvil para dispositivos Android a través de la cual los usuarios podrán guiarse por los diferentes puntos de interés de la localidad que estén visitando.

En este proyecto desarrollo una aplicación que ofrece la posibilidad de hacer el seguimiento de los tours de una forma mas entretenida, por lo tanto los usuarios se entretienen a la vez que descubren la localidad.

El desarrollo se ha llevado a cabo con la finalidad de implementar una aplicación y una API REST con las tecnologías en uso en el sector. Por otra parte hacer uso de una metodología de desarrollo ágil la cual nos permita adaptarnos a los cambios y posibles problemas que nos vayan surgiendo.

Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.

Martin Fowler.

Índice general

Índice de figuras

Índice de Listados

1. Introducción

El turismo representó el 14,6 % del PIB de España en 2018 con un incremento del 2,4 % con respecto al año 2017[1]. Dada la importancia que tiene el turismo en España y la belleza de nuestras ciudades, una buena opción para que el turista disfrute del lugar que visita es enseñándole nuestros mejores sitios. Mediante esta aplicación cualquier persona podrá dar a conocer cuales piensa que son los mejores sitios para visitar, ya sea, una ruta de senderismo por la montaña, o los bares con mejor decoración.

A la hora de visitar una ciudad puede que nos encontremos en la situación de que no tenemos muy claro que visitar o simplemente queremos hacerlo de otra forma. Esta aplicación parte de la idea de dar una forma más entretenida de descubrir una ciudad. Los usuarios tienen la posibilidad de completar uno de estos tours de tres maneras diferentes y a cada cuál con mayor dificultad. Esta aplicación da la posibilidad de conocer nuevas ciudades e incluso partes de tu propia ciudad que de otra forma tal vez no hubieras conocido.

Como cliente tendremos una aplicación para el sistema operativo Android en su versión 6.0 o superior. Me he decidido a realizar una aplicación en Android por el afrontar el desarrollo de un proyecto con una nueva tecnología. Además tiene una gran comunidad de desarrolladores detrás y puedes encontrar guías, documentación y librerías para ayudar al desarrollo, tanto oficiales como otras creadas por el resto de la comunidad. Por otra parte, Android cuenta con una gran tasa del mercado que ronda el 76 % de dispositivos móviles mientras que iOS sólo representa el 22 %.

Y como servidor tendremos una API Rest desarrollada usando el framework de PHP Lumen en su versión 5.7. La elección de este framework de PHP es porque después de haber aprendido Laravel en el Grado, aplicar Lumen es más sencillo ya que es una

[1]Noticia agencia EFE

versión reducida y más rápida de este.

1.1. Objetivos

En este proyecto afronto un gran reto como es el desarrollo de una aplicación nativa para dispositivos Android. Partiendo de no tener ningún conocimiento sobre desarrollo para Android, será necesario realizar una fase de aprendizaje inicial, pasando por la fase de análisis de requisitos de la aplicación hasta llegar al desarrollo de una aplicación funcional.

El objetivo global de este proyecto es crear una aplicación Android, para el usuario final, que utilice una API Rest para obtener los datos. Como objetivo principal está el desarrollo de la aplicación mediante una metodología ágil, gracias a la cual se pueda realizar un desarrollo rápido al cambio y me ayude a gestionar la fase de implementación. Por otra parte, también he tomado como objetivo el aprender a desarrollar una aplicación en Android, ya que desarrollar aplicaciones para dispositivos móviles era uno de los proyectos que tenía pendientes durante el grado.

Con respecto al desarrollo del software, los objetivos específicos son:

- Desarrollar un código limpio, modular y reutilizable.
- Establecer integración continua tanto en cliente como en servidor.
- Realizar más de una release.
- Mantener el código documentado con comentarios.
- Hacer uso de las últimas versiones de librerías y frameworks.

Con respecto a la funcionalidad de la aplicación, los objetivos específicos son:

- Desarrollar una aplicación que muestre un listado de tours guiados disponibles, incluyendo un método para buscar entre los mismos.
- Permitir al usuario realizar uno de estos tours, mostrando indicaciones que le

permitan llegar de una localización a otra del tour realizado.

- Guardar un estado de los tours realizados.
- Desarrollo de las pantallas básicas de la aplicación para la gestión de usuarios.
- Desarrollo de pantallas que permitan la creación y modificación de tours guiados, incluyendo la posibilidad de añadir localizaciones, descripciones de las mismas, así como ordenar las localizaciones.

2. Estado del arte

En esta sección vamos a analizar algunas de las aplicaciones similares que podemos encontrar en el mercado, incluyendo aplicaciones para dispositivos móviles y webs.

Posteriormente veremos como deberíamos enfocar la aplicación para cubrir las funcionalidades de estas aplicaciones que gustan a los usuarios y ver que podríamos mejorar y en que nos diferenciaríamos.

2.1. Geocaching

Geocaching es una aplicación Android creada en el año 2000, en la cuál los usuarios pueden esconder contenedores (caches) en el campo o la ciudad y el resto de usuarios deben encontrarlos. Estos caches contienen un log, por lo que cualquier persona puede esconder un objeto para que los demás usuarios puedan encontrarlos. Los caches deben de incluir un registro, si el tamaño lo permite, en el cual las personas que lo encuentran deben de apuntar su nombre de usuario. Hay diferentes tipos de caches:

- Tradicional: Contenedor que no incluye nada en especial.
- GeoEarth: Es un contenedor tradicional solo que este debe de enseñar algo sobre la naturaleza.
- Mystery: Contenedor tradicional en el que las coordenadas del objeto se deben de descubrir mediante algún tipo de acertijo.
- Multi: Una serie de caches que están enlazados y es necesario encontrar todos para completarlo. Pueden contener partes de un acertijo para resolver unas coordenadas

finales.

2.2. Wikiloc

Wikiloc es una aplicación/web para descubrir y compartir rutas al aire libre. Cada ruta se puede asignar a muchos tipos de deportes, desde senderismo hasta paseo a caballo, pasando por bicicleta de montaña. Cualquier persona puede crear sus propias rutas y seguir otras. Las rutas se pueden clasificar según la distancia, dificultad y tipo de ruta (senderismo, paseo, ciclismo de carretera, ciclismo de montaña, etc).

2.3. Conclusiones

Podemos ver que ya existen una serie de aplicaciones que ofrecen la posibilidad de descubrir sitios nuevos:

- Geocaching: es una aplicación con una comunidad de millones de usuarios y más de tres millones de contenedores que encontrar. Sin embargo, esta aplicación está orientada a la búsqueda de elementos ocultos y resolución de acertijos que pueden llegar a ser complicados.

- Wikiloc: una aplicación con más de doce millones de rutas en todo el mundo y hasta 70 tipos de actividades. El problema es que está orientada al uso durante rutas en montaña.

Viendo los servicios que ofrecen las diferentes aplicaciones, la aplicación a desarrollar va a estar orientada al descubrimiento de sitios de interés ya sea lugares turísticos dentro de una ciudad o en el campo pero siempre con la necesidad de llegar hasta ese sitio para poder completarlo.

3. Metodología

Hacer un desarrollo de calidad no aporta valor directo al producto final, pero si de forma indirecta. Un proceso de desarrollo de calidad aporta reducción en el tiempo de respuesta, cuando hay cambios en las especificaciones o aparecen errores en el sistema, y por lo tanto reducción en los costes del desarrollo.

Para empezar, la metodología a emplear debe de ajustarse a las características del proyecto (especificaciones, tiempo de vida, equipo de desarrollo, ...). Usar una metodología de desarrollo en cascada no es conveniente ya que no se ajusta a la realidad del proceso de desarrollo de software y, cuando se tiene poca práctica, al hacer un análisis completo de la aplicación es muy fácil cometer errores que después implican retrasos en el desarrollo. Por lo tanto se ha decidido utilizar una metodología ágil.

Dentro de las principales metodologías ágiles encontramos Scrum y Kanban. La metodología Scrum no tendría mucho sentido en un proyecto realizado por una sola persona, ya que en Scrum:

- Existen roles para los miembros del equipo.
- Se realizan daily meetings, reuniones diarias en las que se comenta el trabajo realizado el último día y los problemas encontrados.

Por lo tanto la metodología a usar ha sido Kanban, la cual nos ofrece una mayor flexibilidad. Mediante el sistema de issues de GitHub he podido establecer las tareas a realizar y gracias a la opción de Proyectos he podido establecer un tablero Kanban en el cual tener una visión general del estado del mismo (ver sección 4.1.3.4).

3.1. Sistema de control de versiones

Para el sistema de control de versiones he utilizado Git[1]. Gracias al uso de las ramas he podido hacer un desarrollo ordenado con dos ramas de largo recorrido, master y dev.

- master: rama con el código que estaría en producción. Código estable y validado.
- dev: código en desarrollo en el cual se irán añadiendo funcionalidades o mejoras, las cuales están comprobadas, pero no validadas por completo.
- El resto de las ramas partirán y se hará merge siempre en la rama dev. Se crearán para añadir las funcionalidades

A la hora de comenzar una tarea o una serie de tareas que tienen bastante en común, se crea una rama para el desarrollo de estas. La rama debe de partir del último commit de la rama dev y debe de ser mergeada (unida) con esta cuándo su desarrollo se ha finalizado. Si queremos obtener una funcionalidad o una solución a un error, que ha sido unida con la rama dev después de la creación de esta rama, se puede hacer un merge de la rama dev sobre la rama del desarrollo de las tareas.

En la imagen 3.1 podemos ver como sería un merge normal de una rama en la que se está realizando un cambio. Todos los cambios se juntarían en un mismo commit.

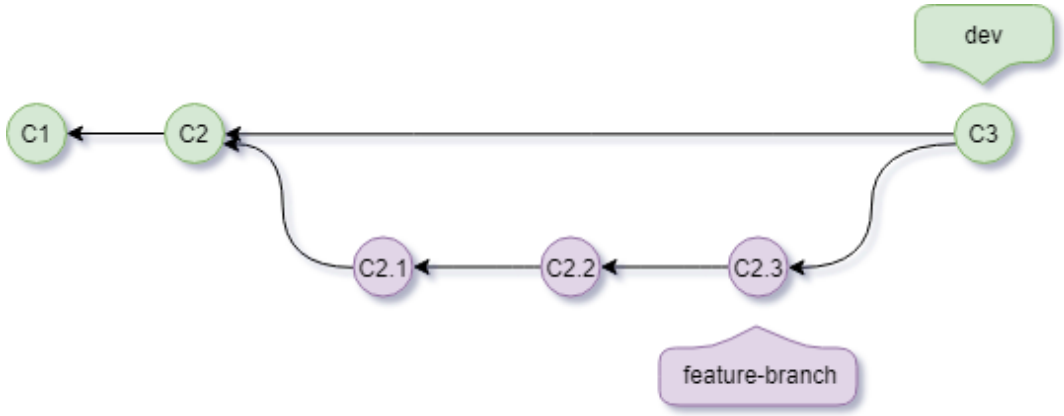

Figura 3.1.: Acción de merge de una rama en la rama dev.

[1]Web oficial de Git. `https://git-scm.com/`

La imagen 3.2 representa la obtención de los cambios nuevos que hay en la rama dev y que queremos en nuestra rama antes de unir con dev, de esta forma podemos comprobar la compatibilidad del código antes de realizar la unión.

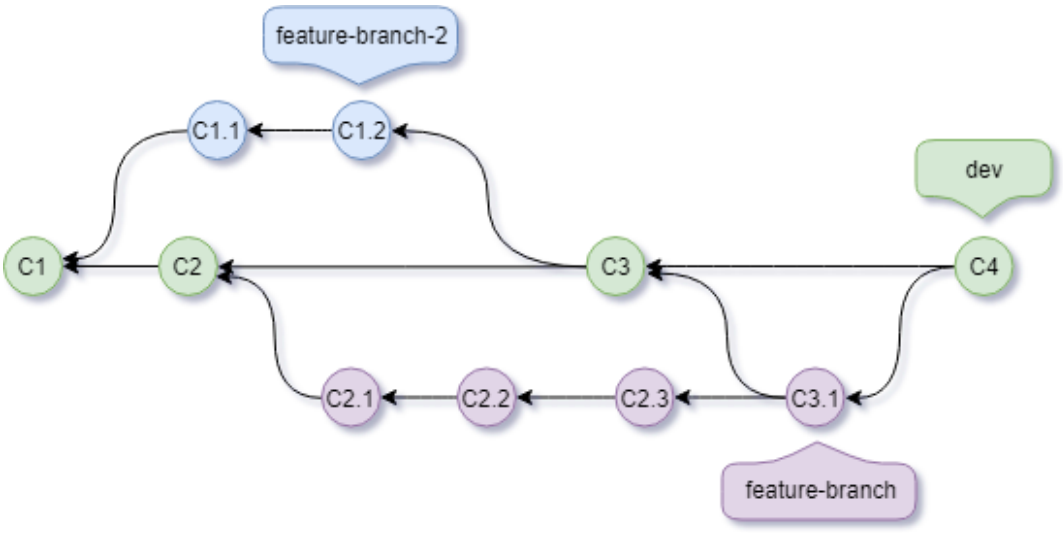

Figura 3.2.: Acción de merge de una rama en la rama dev.

3.2. Issues

El issue es la forma en la que se pueden crear las tareas e incidencias en GitHub y gracias a ellos podemos llevar un seguimiento del trabajo pendiente.

Se pueden crear etiquetas con las que puedes categorizar los issues a tu gusto. En mi caso creé tres etiquetas para diferenciar los issues:

- enhancement: para tareas que pueden añadirse de forma opcional. Son tareas de baja importancia que no aportan mucho valor al producto o incluyen una funcionalidad no prevista (funcionalidades para el futuro).
- features: tareas que añaden funcionalidad y/o valor al producto.
- fix: tareas para arreglar algún fallo. Lo issues con esta etiqueta deben de explicar el problema y la posible solución.

- release: etiqueta para los pull request de dev sobre master. Es decir, para subir el nuevo código desarrollado a la rama de producción.

Al crear el issue, le añadiremos los siguientes elementos:

1. Título descriptivo de la tarea a realizar.

2. Descripción de la tarea con un resumen de lo que se debe hacer. Si es un arreglo, se debería de indicar como replicar el fallo.

3. Persona asignada a resolver el issue. Si fuese un equipo de trabajo, se asignaría al responsable que repartiría las tareas.

4. Etiquetas que categorizan el issue. Se usan varios tipos de etiquetas para facilitar su búsqueda y priorización. (Ver imagen 3.4).

5. Proyecto: indica el tablero donde se van a insertar los issues (ver sección 4.1.3.4).

6. Hito al que pertenece (ver sección 3.3). He usado el nombre de la versión en la que va a ser publicada para poder saber los issues que van a ser resueltos en esa versión.

En la figura 3.3 se puede ver un ejemplo de un issue en el cual se indica el título descriptivo de la tarea a realizar. Debajo encontramos una descripción de la tarea a realizar. Si fuera un fallo encontrado, se deberían de indicar los pasos para poder reproducir el error. En el lado derecho podemos observar el responsable de la tarea, la etiqueta que le ha sido puesta, el proyecto al que ha sido asignado y el hito al que pertenece. Debajo de la descripción se irán añadiendo mensajes con los cambios que va sufriendo el issue, como por ejemplo, un cambio de responsable o el hito al que pertenece. Junto a estos mensajes, se pueden ir añadiendo comentarios que podrían ser necesarios en un futuro.

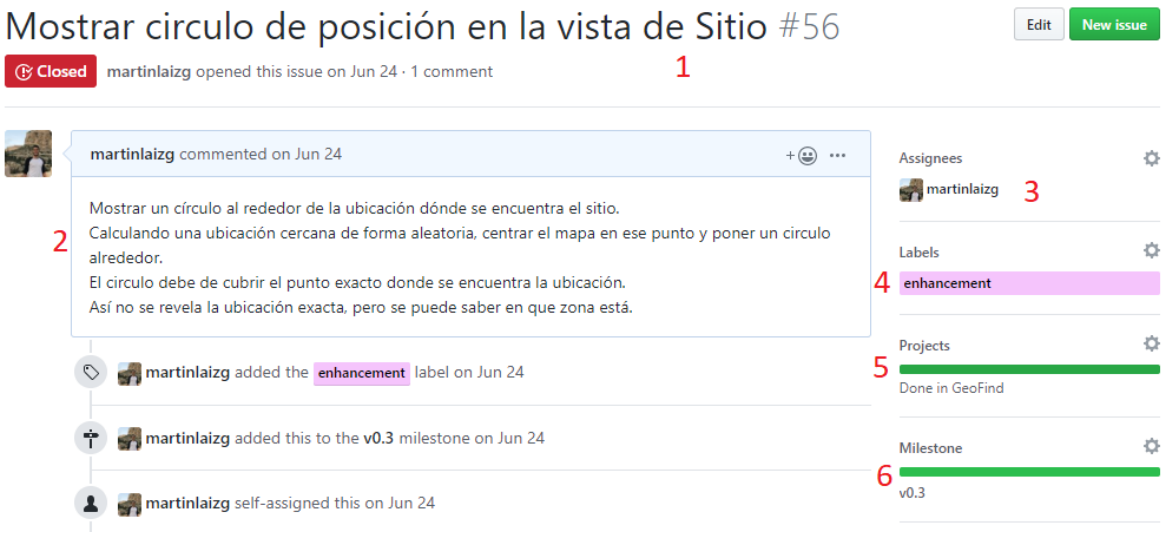

Figura 3.3.: Issue de ejemplo.

En la imagen 3.4 podemos ver como se haría un filtrado. En la parte superior encontramos el filtro aplicado actualmente, el cual está filtrando los elementos que son issues (también podrían ser pull requests), están cerrados y tienen la etiqueta *fix*. Justo al comienzo del listado de issues podemos encontrar una barra en la que podremos filtrar los issues por el autor, las etiquetas que tiene, el proyecto al que pertenece, el hito al que está asignado y la persona que está asignada a realizar la tarea, todo esto con una opción final de ordenado.

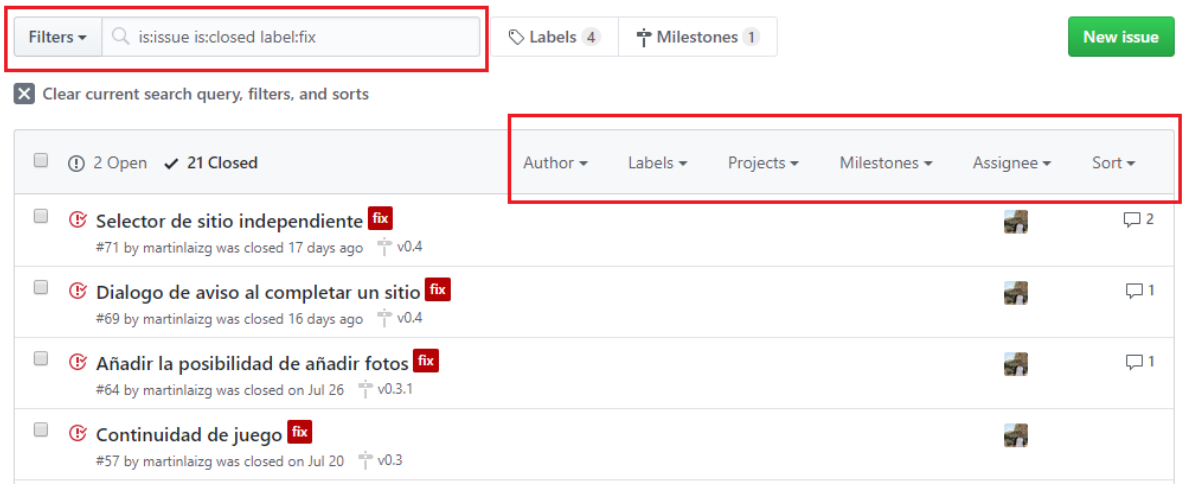

Figura 3.4.: Sistema de filtrado y ordenación de los issues y pull requests de GitHub.

Durante el proceso de gestión de proyecto, voy creando issues y asignándolos a hitos. Esos hitos tienen una fecha marcada para ser completados. En el caso de descubrir que no me va a dar tiempo a completar una tarea, se puede quitar alguna de ellas o cambiarla por otra que consuma menos tiempo. De esta forma se consigue no retrasar la entrega.

Una vez que se va a empezar el desarrollo, se crea la rama y se añade la URL a la rama en un comentario del issue. A medida que se va haciendo el desarrollo se puede ir comentando el problema encontrado, cómo reproducirlo, la posible solución o el resultado esperado.

Para llevar un buen seguimiento de dónde y qué cambios se han hecho, se deben de relacionar los issues con los commits. Para ello a la hora de hacer el commit añadimos al mensaje de este una de las palabras clave de GitHub: close, closes, closed, fix, fixes, fixed, resolve, resolves, resolved y el número del Issue al que queramos referenciar. Por ejemplo: Close #56, para referenciar el issue número 56.

Si a la hora de hacer el commit se nos ha olvidado, añadiremos un comentario al issue con la URL al commit. Por ejemplo: Close URL.

En la imagen 3.5 podemos ver un ejemplo de un comentario a un isssue una vez que se va a empezar el desarrollo. Una vez creada la rama, se añade un comentario al issue indicando el nombre de la rama y un enlace a esta. Justo debajo podemos ver como sería la referencia al issue en el commit. Simplemente añadiendo las palabras *Close #56* nos indica que se le ha referenciado.

Esta referencia también nos ayuda a la hora de hacer los merges, ya que cuando el commit haya sido mergeado con la rama master, este issue pasará a estado cerrado.

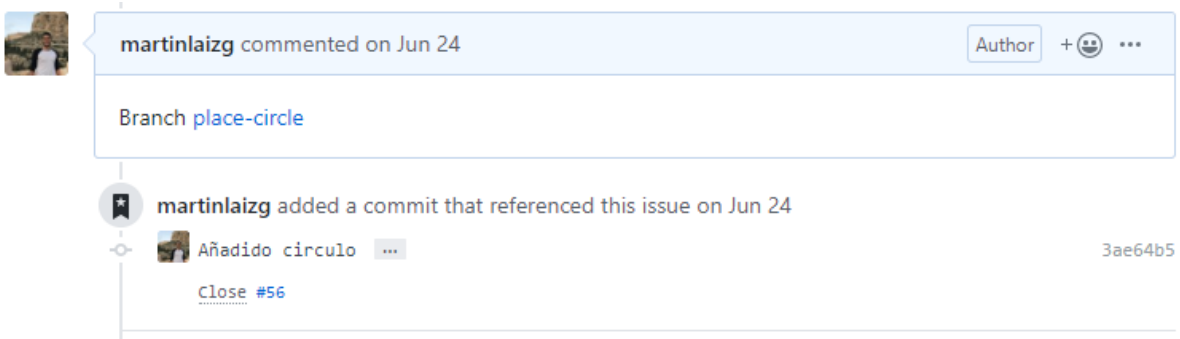

Figura 3.5.: Ejemplo de issue con rama y commit.

3.3. Hitos

Los hitos (Milestones en GitHub) nos ayudan a establecer puntos de control en los que podemos agrupar los issues, indicando qué issues tienen que estar cerrados cuando lleguemos a la fecha del hito.

En este proyecto los he usado para establecer las versiones a entregar. He realizado seis hitos que concuerdan con las seis versiones que he generado. Esta metodología ágil ha permitido el realizar múltiples entregas con los que tener una entrega continua y un progreso constante.

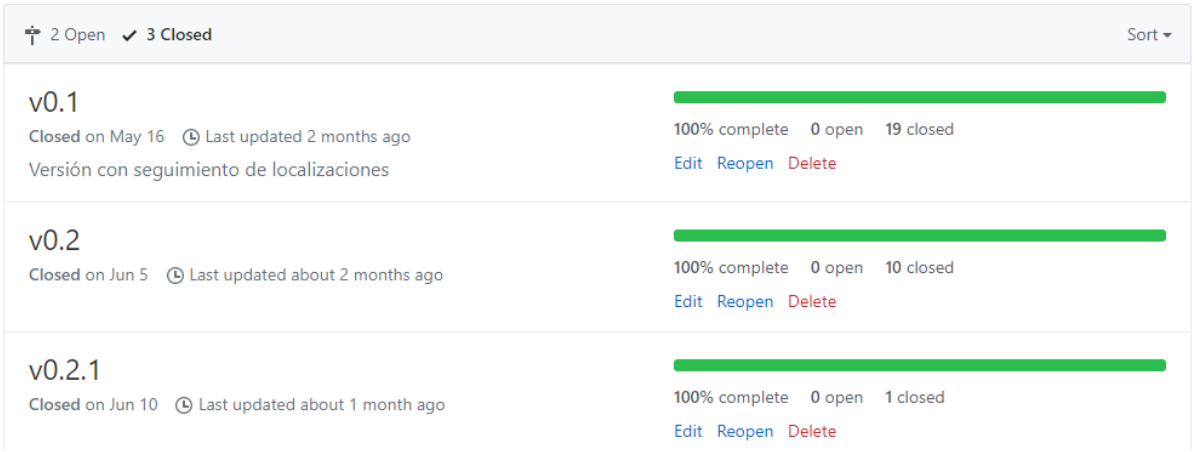

Figura 3.6.: Algunos de los hitos creados durante el desarrollo del proyecto.

3.4. Versionado

Dado el corto período de tiempo, he decidido repartir el proyecto en estas versiones:

- v0.1 (16/05/2019): esta primera versión era una versión de la aplicación que mostraba los tours y los sitios. Además permitía crearlos (pero no modificarlos).
- v0.2 (04/06/2019): en esta segunda versión se añadió el SSO (Single Sign On) y smart lock de Google y se arregló el guiado mediante la brújula y elementos de acceso a la base de datos local del móvil.

    - v0.2.1 (10/06/2019): arreglo de excepción no controlada.

- v0.3 (24/07/2019): llegada esta versión la aplicación ya era funcional. Se arregló la continuidad de guiado al pasar de un sitio al siguiente, se añadieron las preguntas al resolver un tour, se añadieron las imágenes a los tours y se suavizó el movimiento de la brújula. Pero lo más importante, fue añadida la verificación de JWT[2] (JSON Web Token) a las peticiones.

    - v0.3.1 (03/08/2019): arreglo de excepción no controlada.

- v0.4 (26/09/2019): añadido buscador de sitios, diálogos de confirmación al volver atrás de una zona crítica (dónde se pueden perder datos no guardados) cambiado selector de localización y cambiado orden de creación/edición de un tour.

3.5. Integración continua y cobertura de código

Para ir validando de forma automática que todo funciona correctamente, he decidido hacer uso el servicio Travis[3] para integración continua. Tanto en el servidor como el cliente todos los commits en las ramas *master* como en sus ramas *dev* están sujetos a integración continua.

En el caso del cliente, se ejecuta el comando `./gradlew clean build` que borrará los restos de una ejecución anterior y compilará el código fuente. Además lo someterá a un análisis de código donde podemos encontrar fallos graves como podrían ser: falta de traducciones de texto o trozos de código que podrían causar un fallo en tiempo de ejecución. El nivel de detalle del análisis puede ser configurado[4].

La integración continua puede ser configurada para lanzar una serie de test para probar que todo funciona correctamente y generar un informe con la cobertura de código de esos tests. También puede fallar si no se llega a cumplir un porcentaje de cobertura de código[5]. En el caso de fallar la ejecución no se permitiría unir el código a la rama master

[2]Web de JWT. `https://jwt.io/`
[3]Web de Travis. `https://travis-ci.com/`
[4]Guía de Android sobre análisis de código. `https://developer.android.com/studio/write/lint`
[5]Ejemplo de mínimo de cobertura de código. `https://android.jlelse.eu/improving-code-coverage-in-android-app-be8783370c1a`

por la restricción que he establecido sobre esas ramas (ver imagen 3.7).

Mediante la herramienta Codecov[6] se pueden generar gráficos de cobertura de código automáticamente que se añaden automáticamente al commit o pull request que ha lanzado al integración continua (ver imagen 3.8).

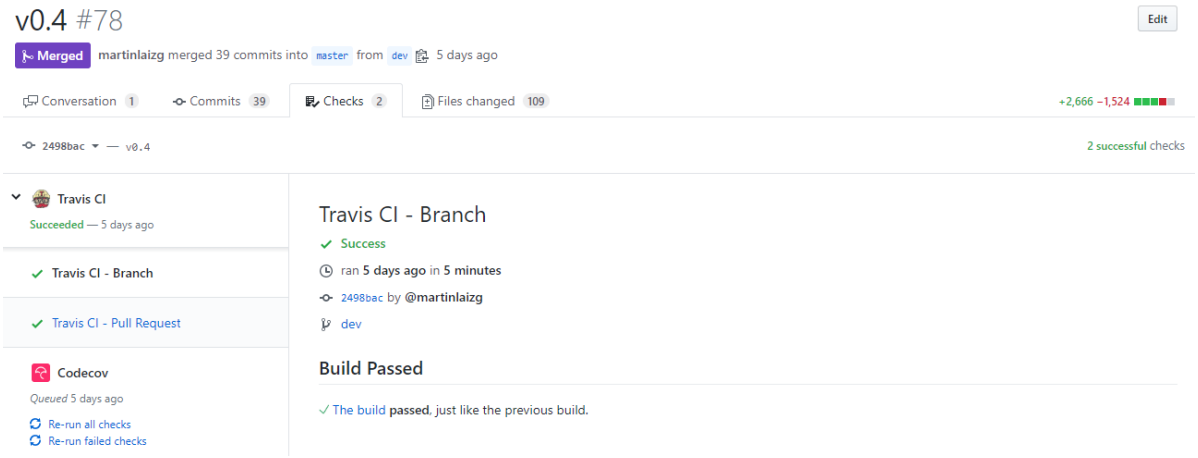

Figura 3.7.: Ejemplo de comprobación de integración continua antes de hace el merge.

[6]Codecov. `https://codecov.io/`

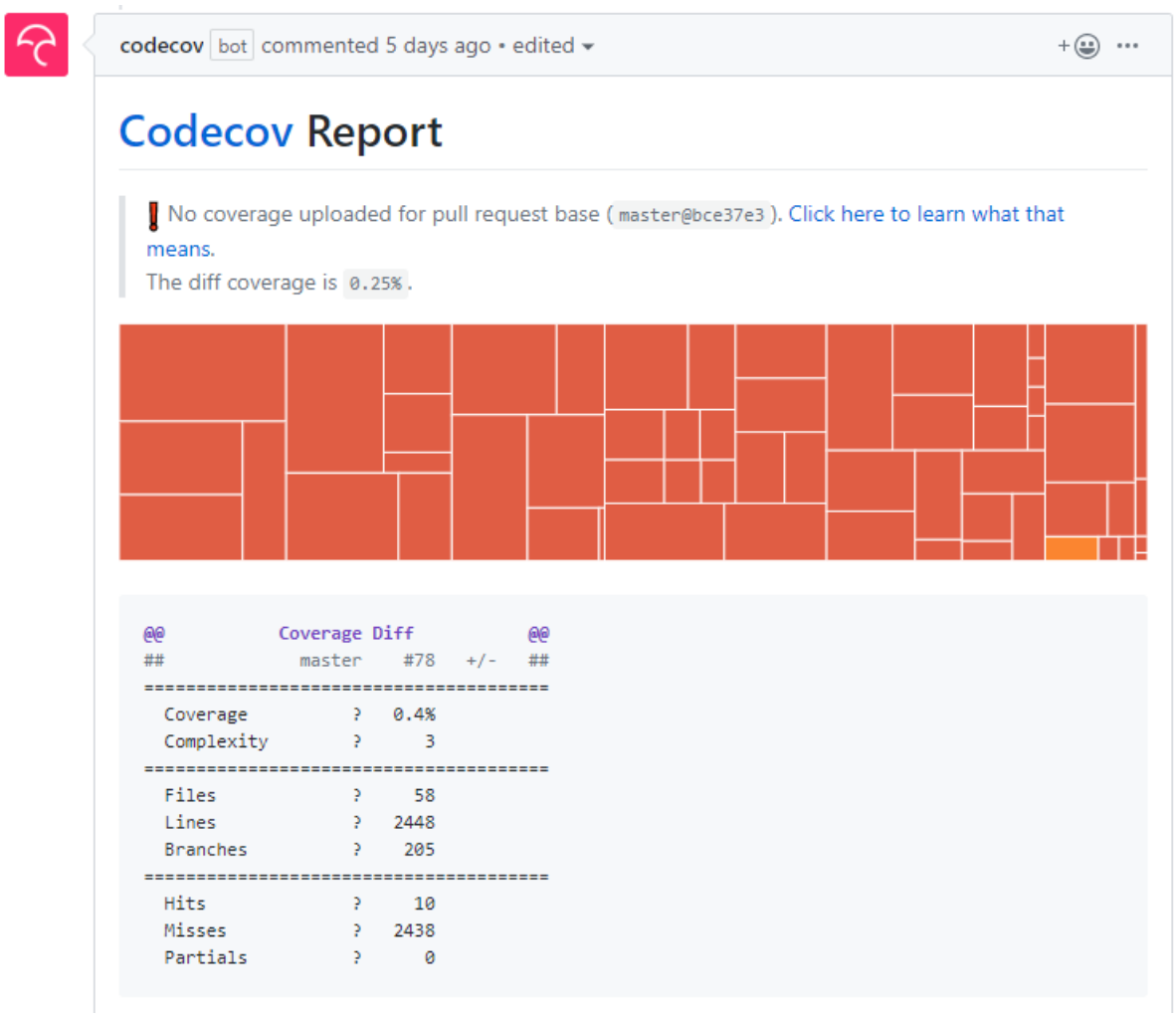

Figura 3.8.: Ejemplo de informe generado por Codecov en GitHub.

El servidor también está sometido a integración continua y a la generación de informes de cobertura, pero dada la limitación de la versión gratuita del servicio Codecov no he podido realizar la visualización en GitHub. Para generar el informe de cobertura ejecutaremos `vendor/bin/phpunit --coverage-clover=coverage.xml` que nos genera el fichero *coverage.xml* con la información necesaria (ver imagen 3.9). Para tener un informe más legible, se puede generar el informe en formato HTML (ver imagen 3.10) con el comando `vendor/bin/phpunit --coverage-html test-coverage-html`. Este comando nos creará una carpeta *test-coverage-html* con la información más legible.

coverage.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<coverage generated="1560853057">
  <project timestamp="1560853057">
    <package name="App\Console">
      <file name="C:\Users\marti\Desktop\geo-find-server\app\Console\Kernel.php">
        <class name="App\Console\Kernel" namespace="App\Console">
          <metrics complexity="1" methods="1" coveredmethods="1" conditionals="0" coveredconditionals="0" statements="1" coveredstatements="1" elements="2"
          coveredelements="2"/>
        </class>
        <line num="25" type="method" name="schedule" visibility="protected" complexity="1" crap="1" count="1"/>
        <line num="28" type="stmt" count="1"/>
        <metrics loc="29" ncloc="14" classes="1" methods="1" coveredmethods="1" conditionals="0" coveredconditionals="0" statements="1" coveredstatements="1" elements="2"
        coveredelements="2"/>
      </file>
    </package>
    <package name="App\Events">
      <file name="C:\Users\marti\Desktop\geo-find-server\app\Events\Event.php">
        <class name="App\Events\Event" namespace="App\Events">
          <metrics complexity="0" methods="0" coveredmethods="0" conditionals="0" coveredconditionals="0" statements="0" coveredstatements="0" elements="0"
          coveredelements="0"/>
        </class>
        <metrics loc="10" ncloc="10" classes="0" methods="0" coveredmethods="0" conditionals="0" coveredconditionals="0" statements="0" coveredstatements="0" elements="0"
        coveredelements="0"/>
      </file>
```

Figura 3.9.: Ejemplo de informe de cobertura XML.

C:\Users\marti\Desktop\geo-find-server\app / (Dashboard)

	Code Coverage								
	Lines			Functions and Methods			Classes and Traits		
Total		0.91%	4 / 438		4.08%	2 / 49		5.00%	1 / 20
Console		100.00%	1 / 1		100.00%	1 / 1		100.00%	1 / 1
Events		0.00%	0 / 1		0.00%	0 / 1		0.00%	0 / 1
Exceptions		0.00%	0 / 8		0.00%	0 / 2		0.00%	0 / 1
Http		0.00%	0 / 406		0.00%	0 / 26		0.00%	0 / 8
Jobs		0.00%	0 / 2		0.00%	0 / 2		0.00%	0 / 1
Listeners		0.00%	0 / 2		0.00%	0 / 2		0.00%	0 / 1
Providers		50.00%	3 / 6		33.33%	1 / 3		0.00%	0 / 2
Place.php		0.00%	0 / 2		0.00%	0 / 2		0.00%	0 / 1
Play.php		0.00%	0 / 3		0.00%	0 / 3		0.00%	0 / 1
Social.php		0.00%	0 / 1		0.00%	0 / 1		0.00%	0 / 1
Ticket.php		n/a	0 / 0		n/a	0 / 0		n/a	0 / 0
Tour.php		0.00%	0 / 3		0.00%	0 / 3		0.00%	0 / 1
User.php		0.00%	0 / 3		0.00%	0 / 3		0.00%	0 / 1

Figura 3.10.: Ejemplo de informe de cobertura HTML.

4. Diseño

Para realizar el primer diseño de la aplicación, he usado el programa Pencil (ver sección 4.1.1.7). Estos diseños me permitieron crear un primer boceto de la interfaz de la aplicación así como analizar los requisitos de la misma.

Nada más abrir la aplicación, el usuario se encontraría con la vista de login y registro.

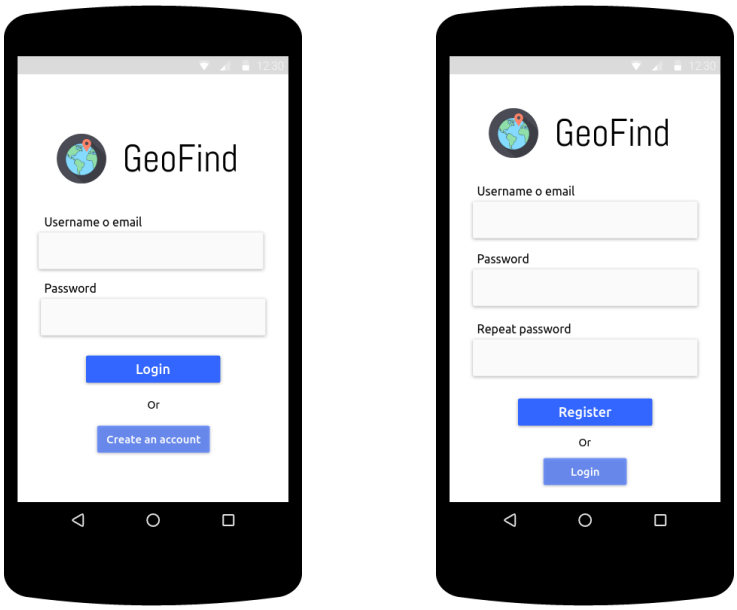

Figura 4.1.: Mockup de login y registro.

Una vez que el usuario entra en la aplicación, se le mostrarán los lugares cercanos relevantes y un buscador.

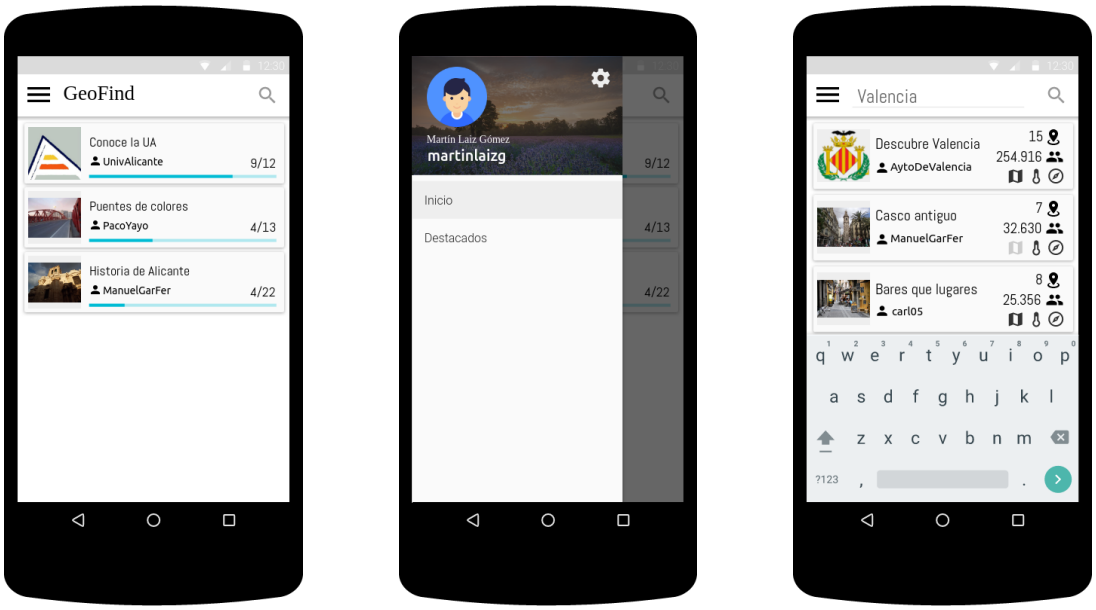

Figura 4.2.: Mockup de lugares cercanos y buscador.

Una vez que el usuario comienza uno de los tours, tendrá tres modalidades: mapa (ver imagen 4.3), brújula (ver imagen 4.4) y termómetro (no tiene mockup).

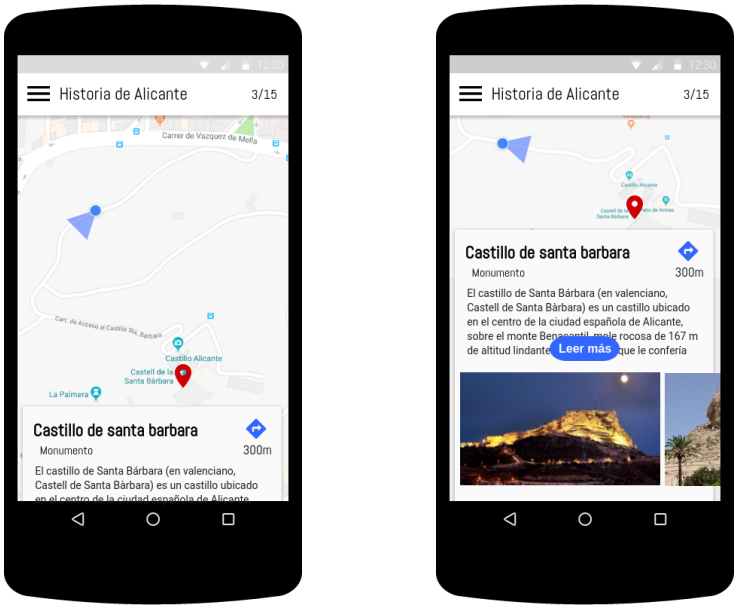

Figura 4.3.: Mockup de tour con mapa.

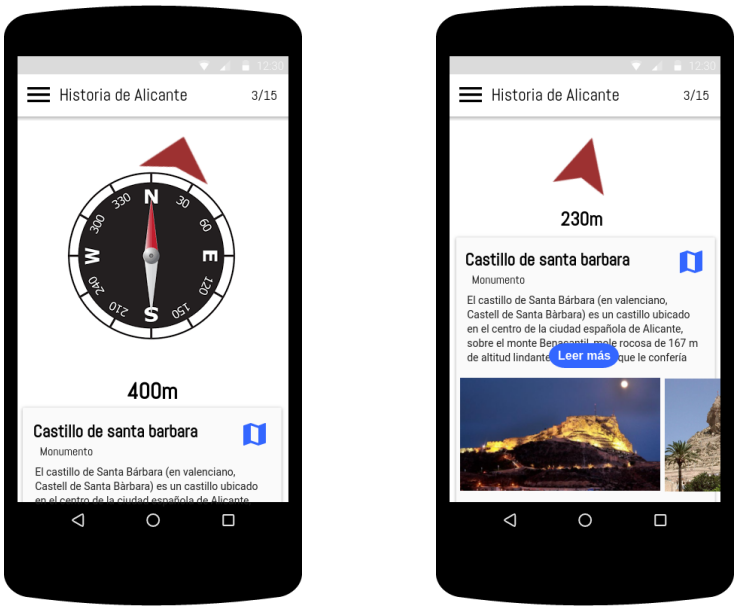

Figura 4.4.: Mockups de tour con brújula.

Una vez que el usuario ha llegado al lugar, se le indicará que ha llegado o, si ese lugar tiene una pregunta, se le mostrará.

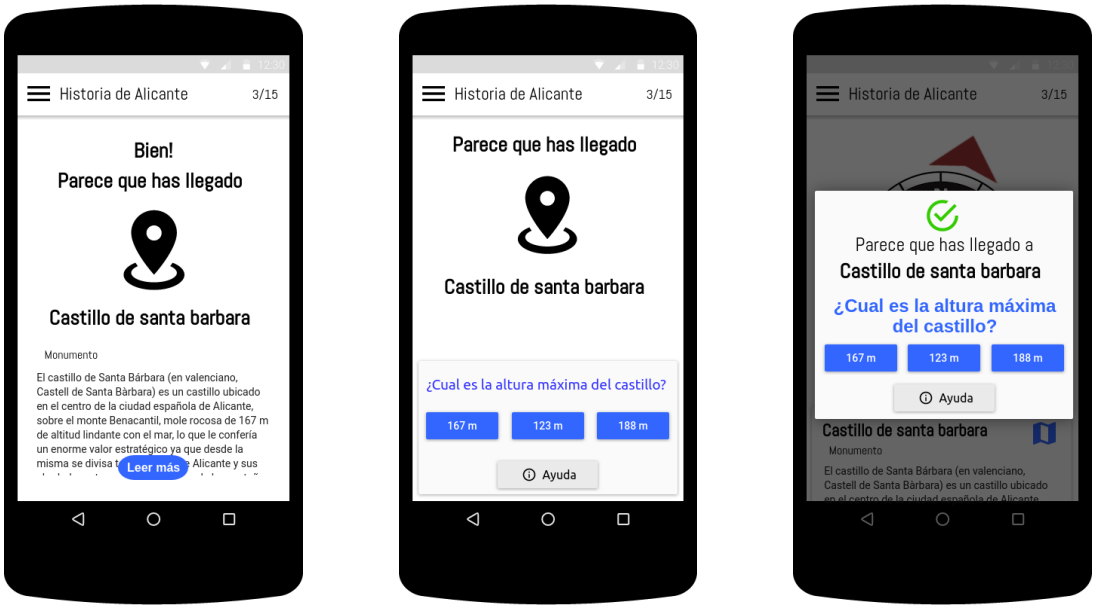

Figura 4.5.: Mockup de lugares cercanos y buscador.

4.1. Tecnologías

En esta sección vamos a presentar la tecnologías utilizadas para el desarrollo del proyecto. En primer lugar se describen las tecnologías utilizadas en el cliente y seguidamente veremos las empleadas en el desarrollo del servidor.

4.1.1. Cliente

La elección de las tecnologías en el cliente han requerido un poco más de trabajo ya que era una tecnología que no había desarrollado antes. Empecé con la idea de desarrollo en web, pero, teniendo en cuenta los conocimientos de mi tutor en el ámbito de Android, decidí decantarme por esta tecnología.

Para la elección de las librerías dentro de la propia aplicación, hice un análisis de las posibilidades que tenía e intenté decantarme por las que parecían ser las más establecidas y que podrían resultar más útiles.

4.1.1.1. Java/Android

El desarrollo de la aplicación se ha realizado utilizando el lenguaje Java, más específicamente la versión 8 para el sistema operativo Android en su versión 6.0 en adelante. Android es el sistema operativo más utilizado en la actualidad, con más de 2.500 millones de dispositivos activos[1]en Mayo de 2019.

He decidido desarrollar la aplicación para la versión Android 6.0 "Marshmallow" o superior, ya que nos da acceso a un 74,8 %[2] de los usuarios del sistema operativo (ver figura 4.6).

[1]Dato en la cuenta de Android en Twitter. `https://twitter.com/Android/status/1125822326183014401`

[2]Datos en la web de Android. `https://developer.android.com/about/dashboards`

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.3%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.3%
4.1.x	Jelly Bean	16	1.2%
4.2.x		17	1.5%
4.3		18	0.5%
4.4	KitKat	19	6.9%
5.0	Lollipop	21	3.0%
5.1		22	11.5%
6.0	Marshmallow	23	16.9%
7.0	Nougat	24	11.4%
7.1		25	7.8%
8.0	Oreo	26	12.9%
8.1		27	15.4%
9	Pie	28	10.4%

Figura 4.6.: Distribución del uso de las versiones de Android.

El sistema operativo Android tiene una documentación muy extensa[3] para el desarrollo de una aplicación funcional. Nos aporta desde buenas practicas en testing, rendimiento o seguridad, hasta guías de diseño para obtener una aplicación con una buena usabilidad. Además cuenta con una larga lista de repositorios con código de ejemplo para muchas de sus librerías[4].

4.1.1.2. Android Studio

Para la implementación del código he decidido utilizar el IDE (Integrated Development Environment, en Español, Entorno de Desarrollo Integrado) Android Studio 3.3, ya que es el entorno de desarrollo oficial de Google para Android.

[3]Guía para desarrolladores. `https://developer.android.com/docs`
[4]Cuenta de Google Samples en GitHub. `https://github.com/googlesamples`

Este está basado en otro IDE, IntelliJ IDEA. [5], uno de los entornos más utilizados para desarrollo Java. IntelliJ IDEA incluye muchas herramientas que nos ayudan a la hora de escribir el código además de ser personalizable para adaptarlo a tus gustos y necesidades.

Google modifica el entorno para añadirle más herramientas, como por ejemplo, la vista de diseño de la interfaz de usuario que se mostrará en la aplicación (ver imagen 4.7) o la ejecución de la aplicación sin necesidad de un dispositivo físico gracias a su emulador.

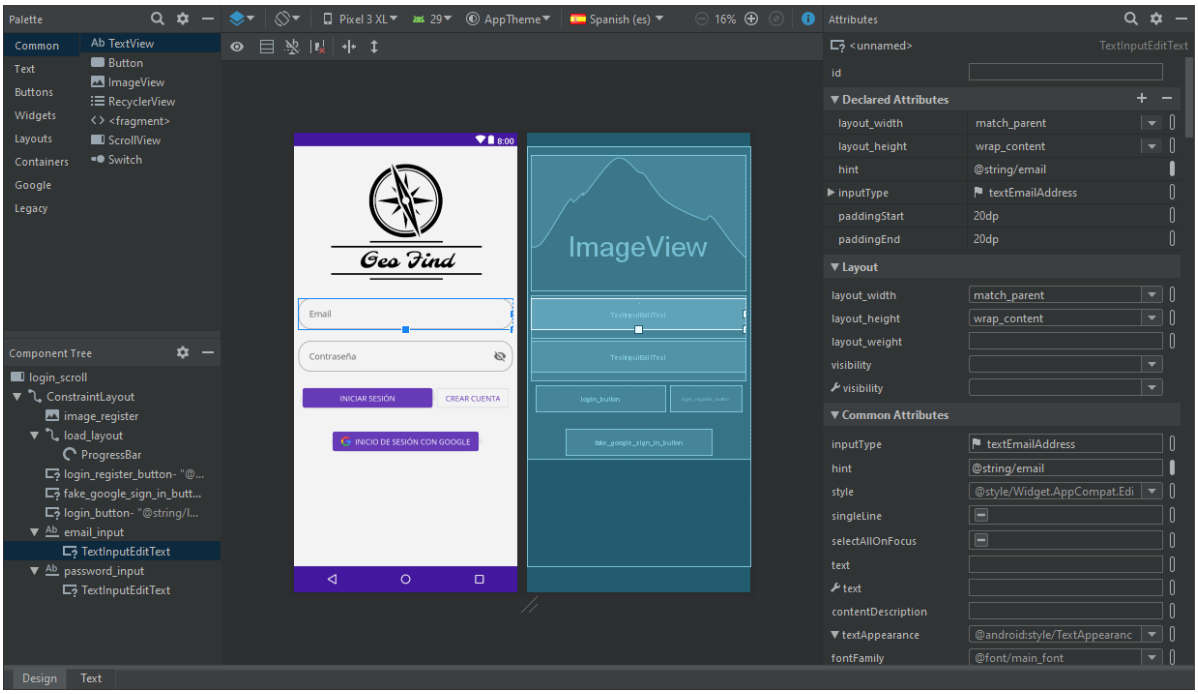

Figura 4.7.: Herramienta de edición de interfaz de usuario en Android Studio.

4.1.1.3. Android Jetpack

Android Jetpack[6] es un conjunto de librerías de Android, publicada en Mayo de 2018 y desarrollada por el propio equipo de desarrollo del sistema operativo.

Los componentes de Android Jetpack son librerías que nos ayudan a abstraernos de una alta complejidad de programación, tales como podría ser la persistencia (ver sección

[5]ItelliJ IDEA `https://www.jetbrains.com/idea/`
[6]Jetpack. `https://developer.android.com/jetpack`

4.1.1.5) o la seguridad[7].

4.1.1.4. Retrofit

Para poder realizar la conexión entre el cliente y la API Rest, he hecho uso de la librería Retrofit[8] para Java que convierte las peticiones a una API Rest en una interfaz de Java y nos devuelve un objeto o una colección de objetos.

Un ejemplo de la definición de la interfaz a un método para realizar llamadas a una API Rest sería este:

```
1 public interface RestClient {
2     @GET("tours/{tour_id}")
3     Call<Tour> getTour(@Path("tour_id") Integer tour_id);
4 }
```

Listado 4.1: Ejemplo de una interfaz de llamada a la API Rest utilizando Retrofit.

La etiqueta o decorador que se pone encima del método (@GET) indica el tipo de petición HTTP a realizar. Actualmente Retrofit soporta cinco métodos[9]: `GET`, `POST`, `PUT`, `DELETE` y `HEAD`. Una llamada a este método (ver listado 4.1) generaría una petición HTTP GET a la ruta "/tours/tour_id". Además indicamos que el parámetro "tour_id" hace referencia a la parte de la ruta que es variable.

A continuación se muestra un ejemplo de la respuesta JSON devuelta por la API para esta petición:

```
1 {
2     "nombre" : "Mejores bares de Alicante (centro)",
3     "descripcion" : "Los mejores bares de la zona centro de Alicante",
4     "numero_localizaciones" : 7,
5     "fecha_creacion" : "2019-03-04T12:34:15.424Z"
6 }
```

Listado 4.2: Ejemplo de la respuesta de la API.

[7]Librería de seguridad. `https://developer.android.com/topic/security/data`
[8]Librería Retrofit. `https://square.github.io/retrofit/`
[9]Documentación Retrofit. `https://square.github.io/retrofit/#api-declaration`

La respuesta en formato JSON sería mapeada automáticamente por Retrofit a una clase Java, la cual tendría la siguiente estructura:

```
public class Tour {
    String nombre;
    String descripcion;
    Integer numero_localizaciones;
    Date fecha_creacion;
}
```

Listado 4.3: Ejemplo de la clase Java que mapearía la respuesta anterior y que nos devolvería Retrofit.

En caso de que la API nos devuelva una lista, simplemente el método deberá devolver "`Call<List<Tour>>`".

4.1.1.5. Room

Room es una de las librerías que forman Android Jetpack. Esta librería está destinada a la creación de una base de datos local. Hace uso de las clases marcadas con la etiqueta `@Entity` para marcar los objetos que formarán una tabla en la base de datos.

Para realizar las consultas a la base de datos se crean unas interfaces con la etiqueta `@Dao`. Estas interfaces contendrán métodos que serán generados automáticamente por la librería. Para realizar las queries de insertado, actualización y borrado se utilizan las etiquetas `@INSERT`, `@UPDATE` y `@DELETE` respectivamente.

Para hacer consultas con una mayor complejidad, se deberá de usar la etiqueta `@QUERY ("SELECT * FROM ...")`, indicando entre comillas la query a ejecutar.

Para parametrizar la consulta, debemos de indicar en la interfaz la variable que queremos inyectar en la consulta y en la query debemos de indicar qué parte es parametrizada con dos puntos y el nombre de la variable, como se muestra a continuación:

```
    @Query("SELECT * FROM tours WHERE id = :tour_id")
        Tour getTour(Integer tour_id);
```

Listado 4.4: Código de ejemplo de un DAO de la librería Room.

4.1.1.6. Butterknife

Para el desarrollo de una aplicación en Android tenemos que declarar la interfaz en un fichero XML y la funcionalidad en código Java. El fichero XML se utiliza para crear el objeto View que formará la pantalla. Si tenemos una entrada de texto en la vista, para poder acceder a este deberíamos de usar el método "findviewById(id)" que nos devolvería el elemento del XML que tenga ese id. Cuando el número de elementos que tenemos que recuperar de la vista se va incrementando, mantener las llamadas a estos métodos llega a "ensuciar" el código.

Para solventar este problema se creó Butterknife[10]. Esta librería permite trasladar estas llamadas del método a la instanciación del objeto que queríamos obtener (ver listados 4.5 y 4.6). Como podemos observar, el código queda más limpio y entendible.

```
1  public class MyFragment extends Fragment {
2      ListView resultList;
3      TextView test;
4      @Override
5      public View onCreateView(LayoutInflater inflater, ViewGroup
           container, Bundle savedInstanceState) {
6          View view = inflater.inflate(R.layout.fragment_my, container,
               false);
7          resultList = (ListView) view.findViewById(R.id.resultListView);
8          test = (TextView) view.findViewById(R.id.textView1);
9          return view;
10     }
```

Listado 4.5: Antes de usar Butterknife.

```
1  public class MyFragment extends Fragment {
2      @Bind(R.id.resultListView) ListView resultList;
3      @Bind(R.id.textView1) TextView test;
4      @Override
5      public View onCreateView(LayoutInflater inflater, ViewGroup
           container, Bundle savedInstanceState) {
6          View view = inflater.inflate(R.layout.fragment_my, container,
               false);
7          ButterKnife.bind(this, view);
8          return view;
9      }
```

Listado 4.6: Después de usar Butterknife.

[10]Butterknife en GitHub. `https://jakewharton.github.io/butterknife/`

4.1.1.7. Pencil

A la hora de diseñar la interfaz es recomendable utilizar alguna herramienta para realizar wireframes (diseños básicos) y cuando la idea ya se ha establecido, crear mockups para que el cliente valide que el diseño le gusta. Estos diseños nos ayudan a encontrar posibles problemas en la fase de diseño que podrían surgir más adelante.

Para este proyecto he decido utilizar Pencil[11]. Esta herramienta tiene una serie de librerías[12] para crear diseños, pero al llevar 2 años sin mejoras, esos elementos han quedado un poco desfasados. Sin embargo, gracias a la comunidad, se pueden añadir otros más modernos y con más detalle[13]. Yo he utilizado uno que aporta el diseño de Android Lollipop[14] y otro que trae iconos con diseño Material Design [15].

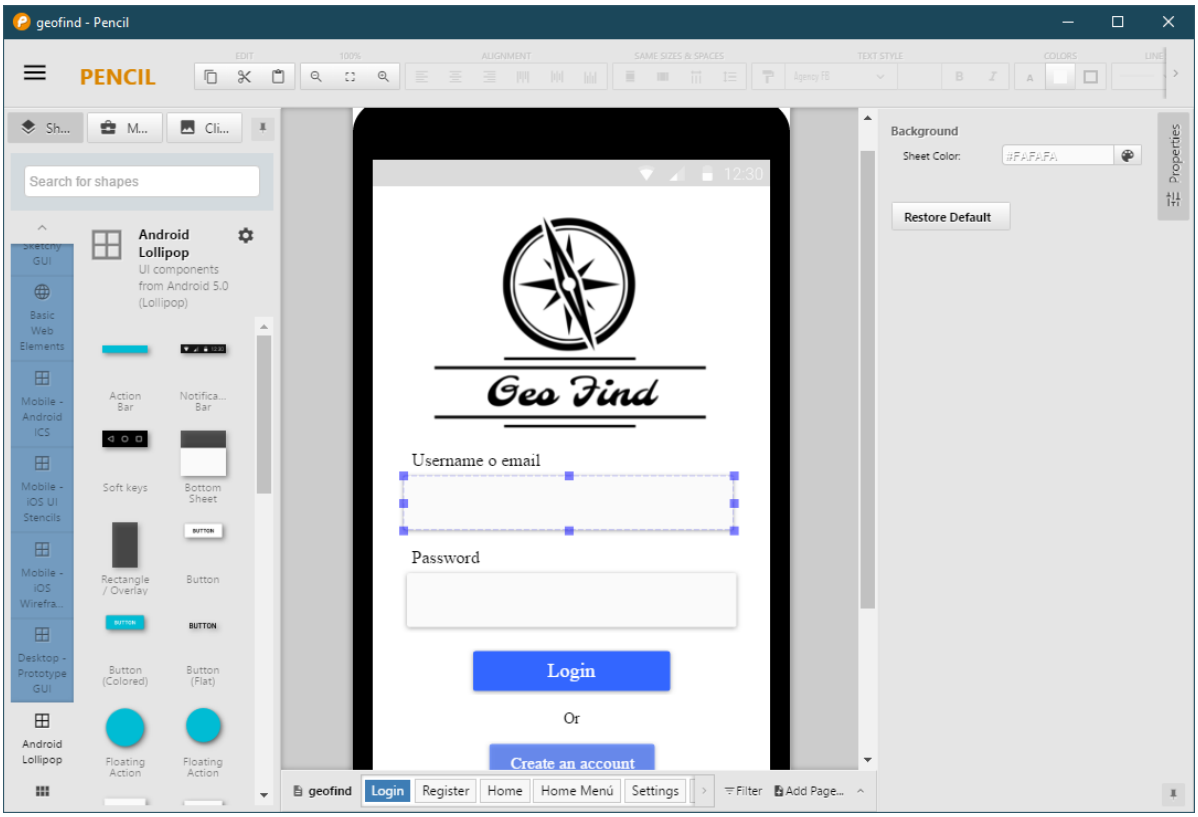

Figura 4.8.: Aplicación de diseño de mockups Pencil.

[11] Web de Pencil. `https://pencil.evolus.vn/`
[12] Elementos predefinidos. `https://pencil.evolus.vn/Stencils-Templates.html`
[13] Mas elementos para Pencil. `https://nathanielw.github.io/pencil-stencils/`
[14] Android Lollipop. `https://www.android.com/intl/es_es/versions/lollipop-5-0/`
[15] Guias de diseño Material Design. `https://material.io/design/`

4.1.2. Servidor

A la hora de seleccionar el stack tecnológico para utilizar en el servidor, ya tenía la idea de utilizar Lumen, un framework PHP para desarrollo de API Rest. Lumen es la versión ligera de Laravel, que este a su vez está basado en Symfony [16]. Por lo tanto después de aprender a utilizar Laravel, en la asignatura Diseño de Sistemas Software, y ver la cantidad de cosas que se podían hacer con este, no tenía la menor duda de que quería hacer una API Rest con este framework.

4.1.2.1. PHP/Lumen

Lumen[17] es un framework PHP con el cual podemos crear API Rest de una forma rápida y sencilla. Lo bueno que tiene Lumen es que es rápido y si necesitas alguna de las funcionalidades de Laravel, es fácil añadirla sin tener que cargar con todo lo que trae Laravel de serie.

Partiendo de la base de Lumen, decidí añadirle el módulo Eloquent de Laravel. Este módulo nos abstrae de la complejidad de acceso a base de datos realizando un Mapeo objeto-relacional (Object-Relational mapping o ORM), por lo tanto podemos tratar entidades de base de datos cómo si fueran objetos.

Si no hiciéramos uso de este módulo, deberíamos de tratar con las consultas y resultados de base de datos. Gracias a este módulo podemos realizar consultas como esta:

```
1       // SELECT * FROM tours WHERE id = 1;
2   Tour::find(1);
3       // SELECT * FROM tours WHERE titlulo like '%Alicante%'
4   Tour::where('titulo','like','%Alicante%')->get();
```

Listado 4.7: Diferencia al usar Eloquent.

Como se puede observar, este modulo es muy intuitivo y fácil de usar, por eso decidí añadirlo a Lumen.

[16]Web de Symfony. https://symfony.com/
[17]Web de Lumen. https://lumen.laravel.com/docs/5.7

4.1.2.2. Visual Studio Code

Visual Studio Code[18] es un editor de texto ligero y potente creado por Microsoft en Noviembre de 2015. Dispone de muchas extensiones[19] con las cuales añadir mejoras al editor, como puede ser una herramienta que borre la base de datos y lance las migraciones de Lumen (ver sección 6.1.4).

4.1.2.3. Heroku

Para poder acceder a la API desde el cliente, ésta debe estar alojada en algún servidor. Para ello he decidido utilizar el servicio de Heroku[20]. Este servicio nos permite alojar aplicaciones web o, como es mi caso, una API Rest, y vincularlo con la cuenta de GitHub (ver sección 4.1.3.4). Decidí usar la rama master como la rama a tener publicada en Heroku, de forma que cada vez que se hace un commit sobre esta rama, se lanza el despliegue en Heroku solo si se ha completado la integración continua (ver sección 3.5 e imágenes 4.10 y 4.9).

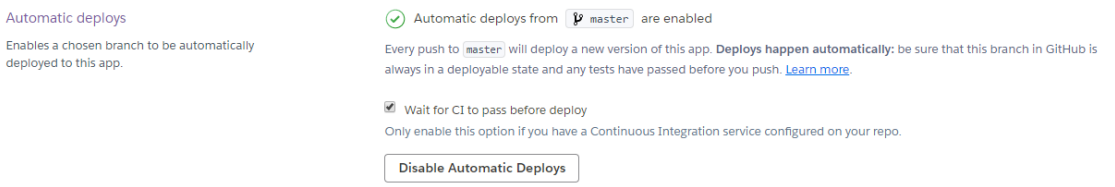

Figura 4.9.: Ejemplo de un despliegue lanzado automáticamente desde la última versión de la rama master de GitHub.

[18]Web de Visual Studio Code. `https://code.visualstudio.com/`
[19]Tienda de extensiones. `https://marketplace.visualstudio.com/`
[20]Heroku. `https://www.heroku.com/`

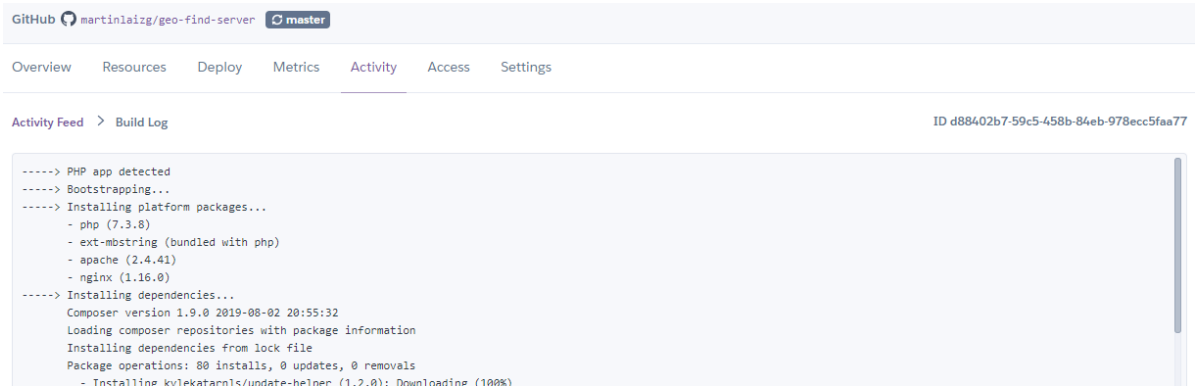

Figura 4.10.: Ejemplo de un despliegue desde la última versión de la rama master de GitHub.

4.1.2.4. RAML (RESTful API Modeling Language)

Para consumir una API REST se necesita saber qué peticiones acepta y cuáles son sus parámetros, para que el desarrollador del cliente sepa las peticiones que puede realizar. Para esto voy a utilizar RAML (RESTful API Modeling Language o Lenguaje de Modelado de API RESTful)[21].

RAML es un lenguaje de modelado de datos que utiliza el formato YAML[22], que nos permite definir todas las propiedades que tendrá la petición HTTP.

Gracias al uso de este modelado de datos, a partir de un fichero en formato RAML se puede generar una página web con la herramienta raml2html[23].

4.1.3. Comunes

En los siguientes apartados veremos las herramientas que he utilizado y que son comunes a ambas partes del proyecto.

[21] Web de RAML. `https://raml.org/`
[22] YAML en la Wikipedia. `https://es.wikipedia.org/wiki/YAML`
[23] raml2html en GitHub. `https://github.com/raml2html/raml2html`

4.1.3.1. JWT (JSON Web Token)

Para validar el usuario que realiza la petición, he decidido usar JWT[24]. JWT nos permite transmitir información sobre el usuario y verificar que no ha sido modificada. El token está formado por tres textos en formato JSON y codificados en Base64URL[25], separados por un punto.

1. Cabecera: la primera parte es la cabecera que por defecto solo incluye el tipo de token y el algoritmo usado para la firma.

2. Cuerpo: la segunda parte es el cuerpo con la información que queremos transmitir.

3. Firma: para finalizar está la firma, que es la concatenación de la cabecera y el cuerpo separado por un punto y cifrado con una contraseña utilizando el método indicado en la cabecera.

Dado que el texto no va cifrado, se debe te tener cuidado para qué se utiliza. En este caso se utilizará como token para el usuario cuando realice el login y será enviado en la cabecera en todas las peticiones que haga. De esta forma podremos verificar que es la persona que ha iniciado sesión y no una suplantación de identidad.

[24]Web de JWT. `https://jwt.io/`
[25]Base64 en Wikipedia. `https://es.wikipedia.org/wiki/Base64`

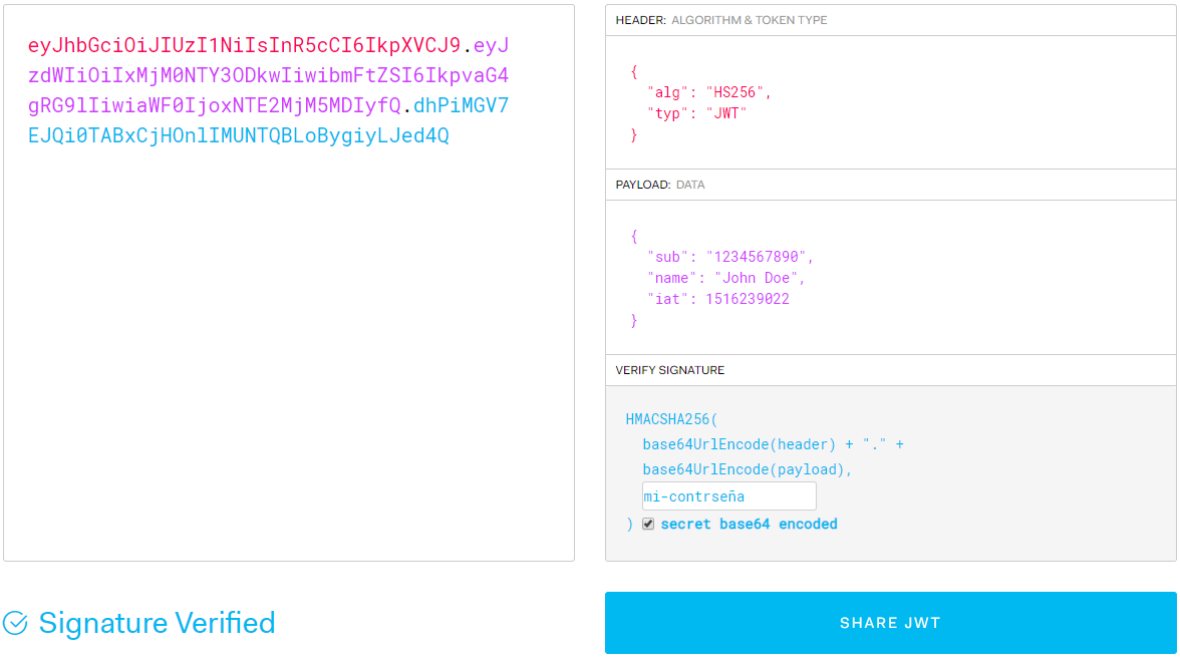

Figura 4.11.: Ejemplo de generación de JWT.

4.1.3.2. Google SSO

Para facilitar el inicio de sesión del usuario, he decidido hacer uso del SSO (Single Sign On) de Google[26].

Cuando el usuario pulsa el botón de iniciar sesión con la cuenta de Google (ver imagen 4.12) se le preguntará por la cuenta que desea usar. En ese momento se obtiene un token JWT, además de otra información que habremos indicado que queremos al registrar nuestra aplicación en Google API Console[27].

Mandaremos ese token al servidor, validaremos la información en los servicios de Google[28] y obtendremos los datos necesarios para crear la cuenta o iniciar sesión sin usar contraseña.

[26]SSO Google en Android. `https://developers.google.com/identity/sign-in/android/start-integrating`
[27]Google API Console. `https://console.cloud.google.com/apis`
[28]Verificar SSO Google. `https://developers.google.com/identity/sign-in/android/backend-auth#verify-the-integrity-of-the-id-token`

Figura 4.12.: Botón de inicio de sesión con Google.

4.1.3.3. Draw.io

Durante la fase de análisis inicial, y durante la fase de análisis de cada tarea, es necesario una herramienta con la que crear diagramas. Estos nos servirán para tener claro qué se va a hacer, y así descubrir los posibles problemas que puedan surgir, antes de comenzar con la implementación.

La herramienta que he decidido utilizar ha sido Draw.io[29]. Las principales funcionalidades que tiene son: una gran librería de figuras, los elementos que forman los diagramas son personalizables, permite almacenar los diagramas en diferentes servicios online (ver imagen 4.14) y modificarlos entre varias personas de forma simultánea.

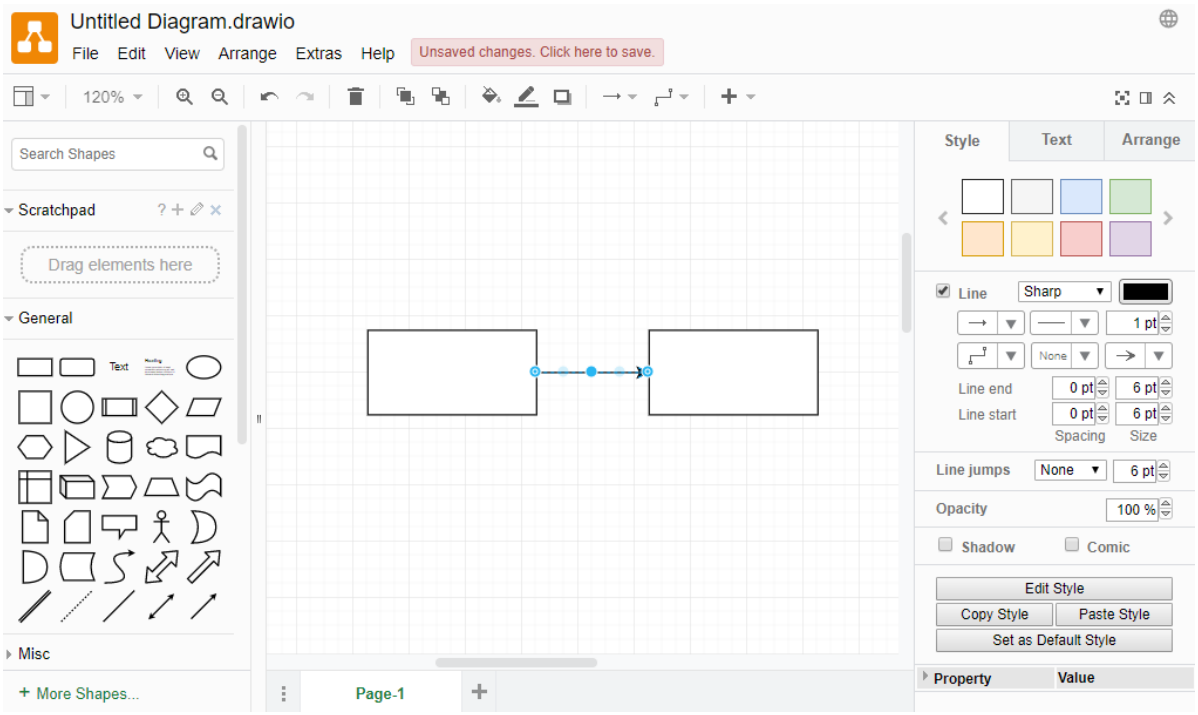

Figura 4.13.: Herramienta de diagramas Draw.io.

[29]Web de Draw.io. `https://www.draw.io/`

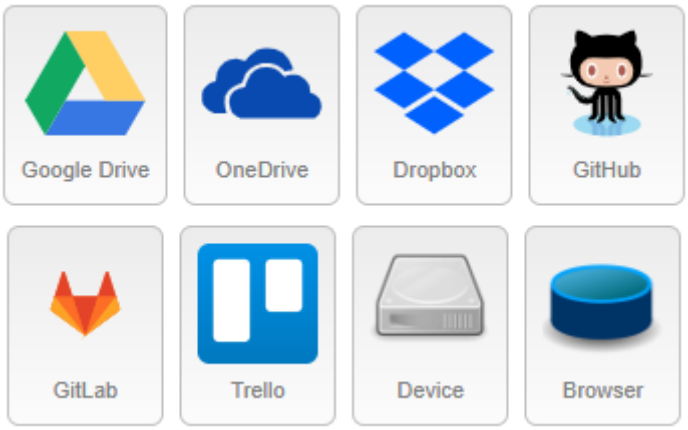

Figura 4.14.: Servicios de almacenamiento para diagramas de Draw.io.

4.1.3.4. GitHub

Para mantener el sistema de control de versiones con Git en un repositorio remoto he decidido utilizar el servicio GitHub. Con GitHub podemos mantener una copia en remoto de seguridad además de poder compartirla entre diferentes ordenadores. Me decidí por este servicio por la integración que tiene con otras herramientas como son Travis y Codecov (ver sección 3.5), además de tener gratis repositorios privados ilimitados[30].

A través de la web de GitHub[31] podemos acceder a una sección donde tendremos un tablero Kanban (ver imagen 4.15) que nos ayuda a tener una imagen del estado actual del proyecto (ver sección 3). Gracias a la automatización (ver imagen 4.16) del tablero, que incorpora GitHub, se van añadiendo de forma automática todos los issues nuevos (ver sección 3.2) en la columna "To do". Se moverán de forma manual los issues a la columna "In progress" cuando se comience con su desarrollo y a "Done" cuando se haya completado.

[30]Precios GitHub. `https://github.com/pricing`
[31]Web de GitHub. `https://github.com`

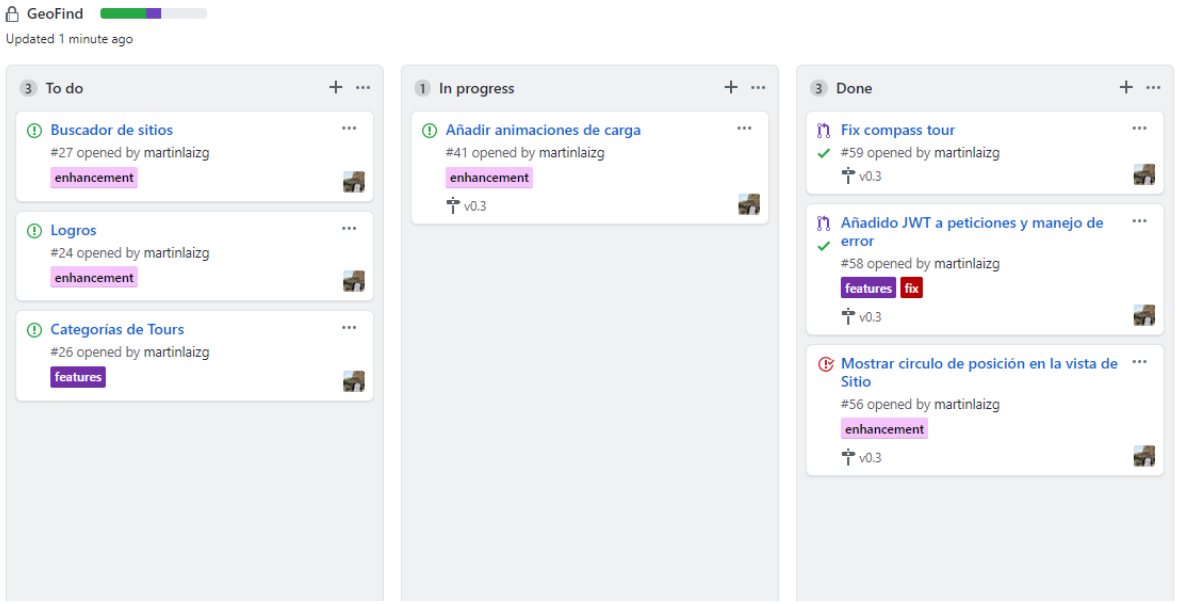

Figura 4.15.: Tablero Kanban en GitHub.

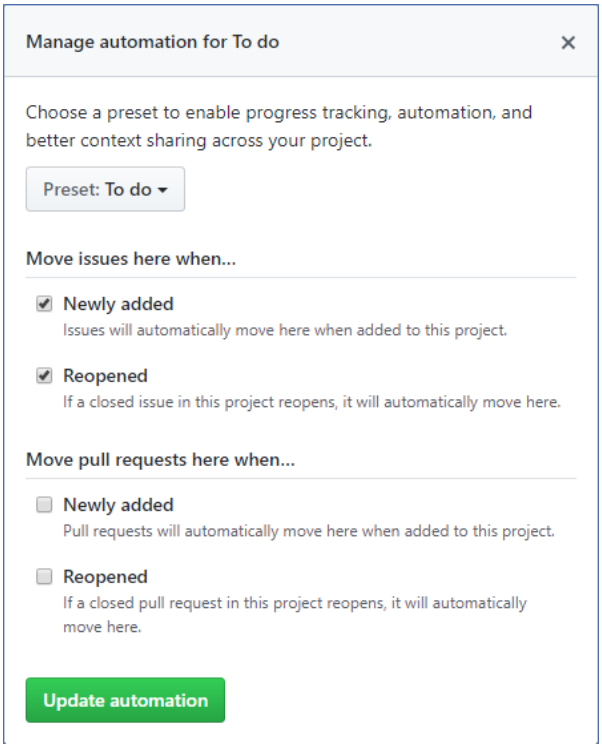

Figura 4.16.: Reglas por las que serán añadidos los issues a esa columna.

Otra de las funcionalidades que tiene GitHub es la publicación de versiones. A través

de esta publicación podemos añadir una descripción con las funcionalidades añadidas o los errores corregidos. En el momento que se crea la publicación, se enlaza con un commit y se crean dos ficheros comprimidos con el código fuente en ese commit. Además del código fuente, se puede añadir otros elementos a la publicación, como podría ser el fichero APK generado para instalar en Android.

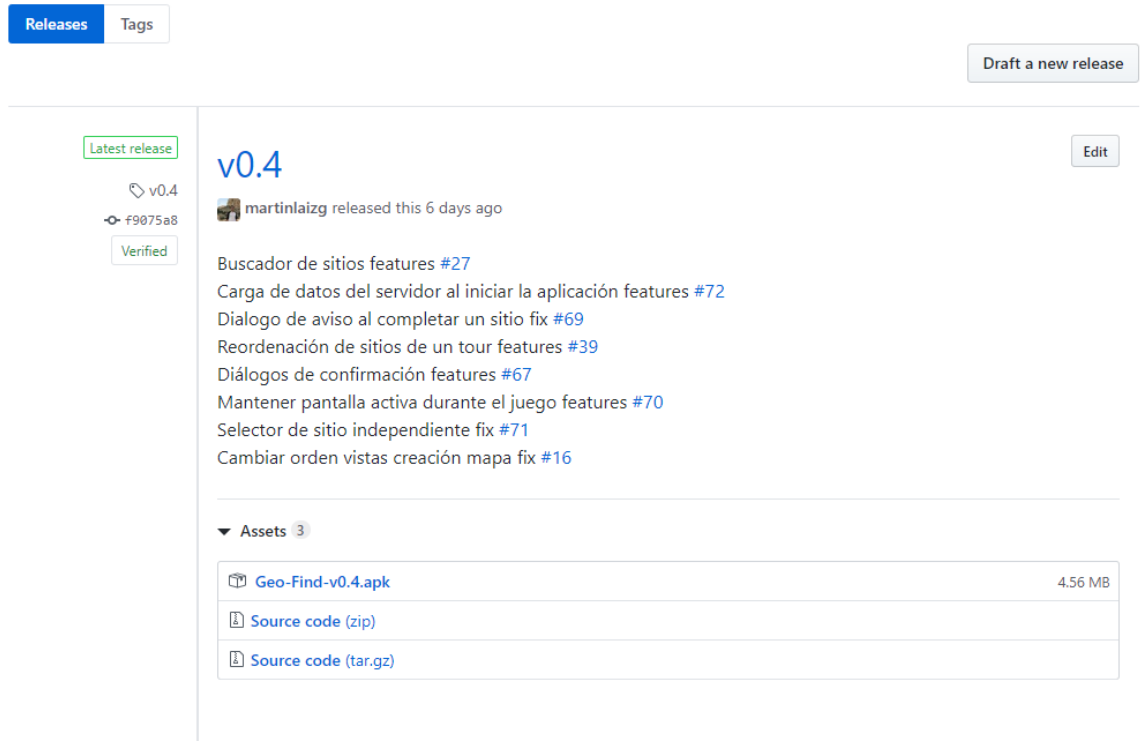

Figura 4.17.: Sección de publicaciones.

A la hora de unir dos ramas de nuestro sistema de control de versiones (ver sección 3.1) se puede hacer mediante los pull requests (solicitudes de unión). Estas solicitudes permiten tener una versión del código en la que está unido el código pero no se ha subido aún, por lo tanto se puede realizar la integración continua (ver sección 3.5) y verificar que todo funciona correctamente.

4.1.3.5. GitKraken

Realizar la gestión de Git a través de línea de comandos puede llegar a ser tedioso por lo que me decidí a utilizar un programa que me ayudara a ello a través de una interfaz.

GitKraken fue creada en Agosto de 2014 y ya cuenta con un millón y medio de usuarios. Algunas de las funcionalidades por las que me decanté a usar GitKraken fueron: una interfaz intuitiva, la integración con GitHub y que es multiplataforma (Windows, Linux y MacOS).

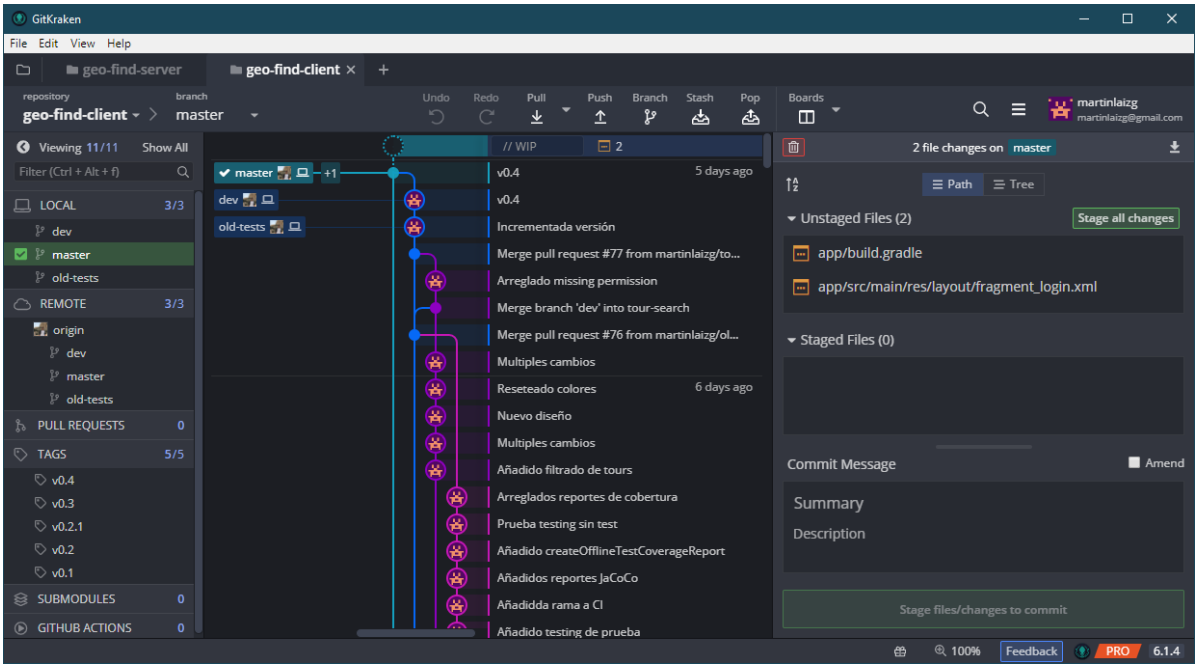

Figura 4.18.: Aplicación de GitKraken en Windows 10.

5. Arquitectura

Establecer una arquitectura clara de acceso a datos es importante en el desarrollo. Facilitará la escalabilidad del proyecto y además nos puede ahorrar tiempo de desarrollo, errores, posibles refactorizaciones de código futuras, así como duplicación de código que harían el proyecto insostenible.

Primero veremos la arquitectura utilizada en el servidor que viene condicionada por el framework usado y a continuación la utilizada en el cliente, que al no tener una estructura definida, deberemos de diseñarla nosotros como mejor nos convenga.

5.1. Servidor

Al usar el framework Lumen, estamos condicionados a la arquitectura que nos define el propio framework. En este caso Laravel define la arquitectura MVC (Modelo-Vista-Controlador) pero al usar Lumen, destinado al desarrollo de una API, la vista se elimina, por lo tanto nos quedarían solamente el modelo y el controlador.

En el modelo se definen las entidades de base de datos y cómo se relacionan entre ellas, y el controlador es el que consulta los modelos y crea la respuesta.

El esquema de base de datos que nos quedaría sería el siguiente.

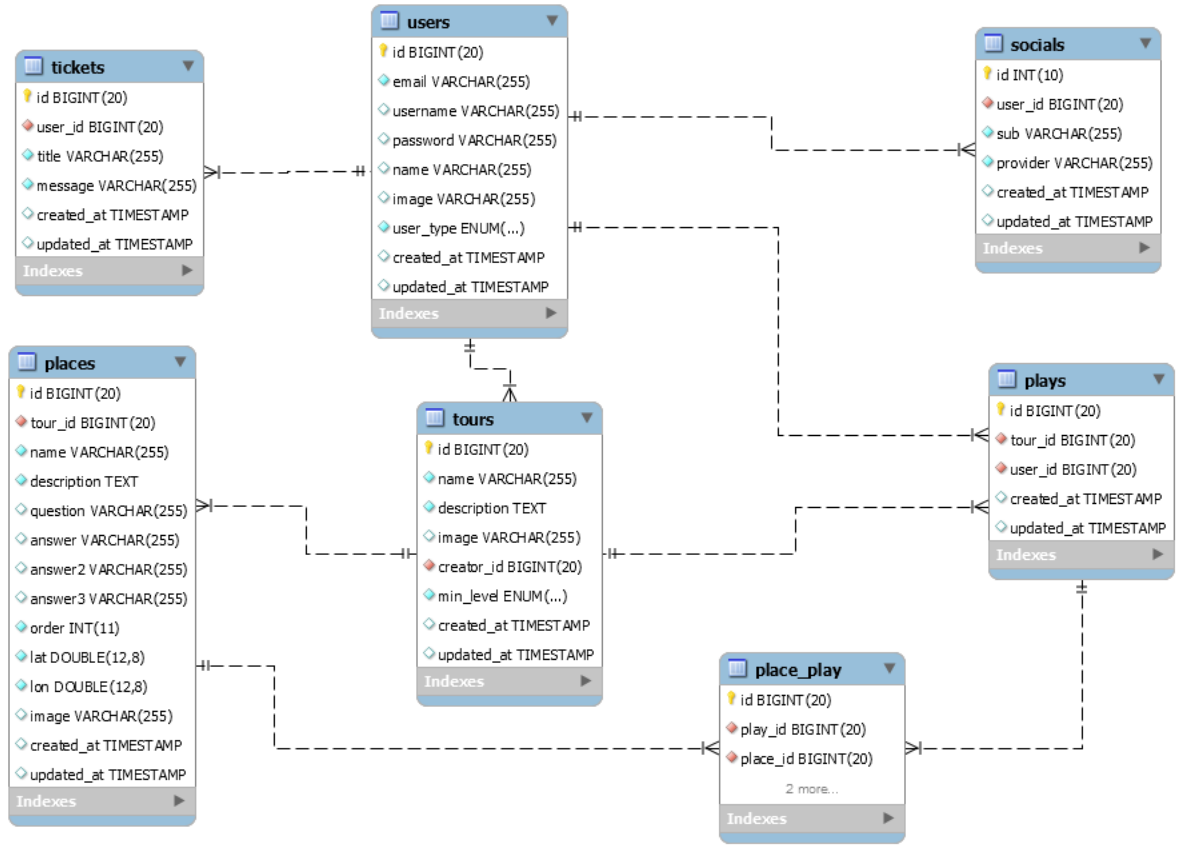

Figura 5.1.: Esquema de base de datos en el servidor.

5.2. Cliente

Para la implementación de la aplicación se ha decidido la implementación de la arquitectura MVVM (Model-View-ViewModel) (ver imagen 5.2).

Los elementos de esta arquitectura son:

- Vista: la vista de la aplicación está formada por “Fragments”, los cuales manejarán toda la lógica de la interfaz y realizarán llamadas asíncronas al ViewModel.

- ViewModel: el viewModel es el encargado de atender las llamadas de la vista y devolver los datos de forma asíncrona. Una vista debe tener un viewModel pero puede ser compartido entre varias. La comunicación con la vista se realiza con

objetos LiveData (ver sección 6.2.3).

- Repositorio: los viewModels harán llamadas a los repositorios para obtener los datos de la base de datos local o de la API de forma abstracta. Los repositorios son los encargados de obtener los datos de la base de datos (Room) o de la API (Retrofit) y tratarlos para ser devueltos al viewModel.

- Modelo (Room): para el acceso a la base de datos local se utiliza Room (ver sección 4.1.1.5) a través de las interfaces DAO.

- Servicio Remoto (Retrofit): el acceso a la API es controlado por la librería Retrofit, para la cual declaramos una interfaz donde establecemos las llamadas que se pueden hacer y cómo se deben hacer (ver sección 4.1.1.4).

En la siguiente imagen podemos ver cómo están estructurados los elementos comentados anteriormente.

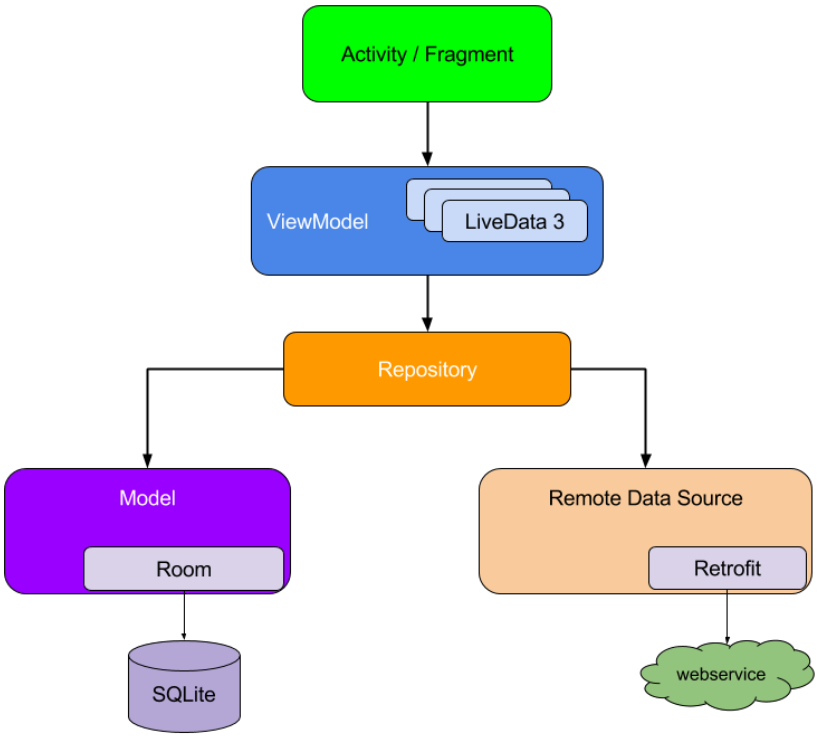

Figura 5.2.: Arquitectura seguida en Android.

En el siguiente diagrama de clases (ver imagen 5.3), podemos ver cómo están divididas las capas de la aplicación y cómo se relacionan.

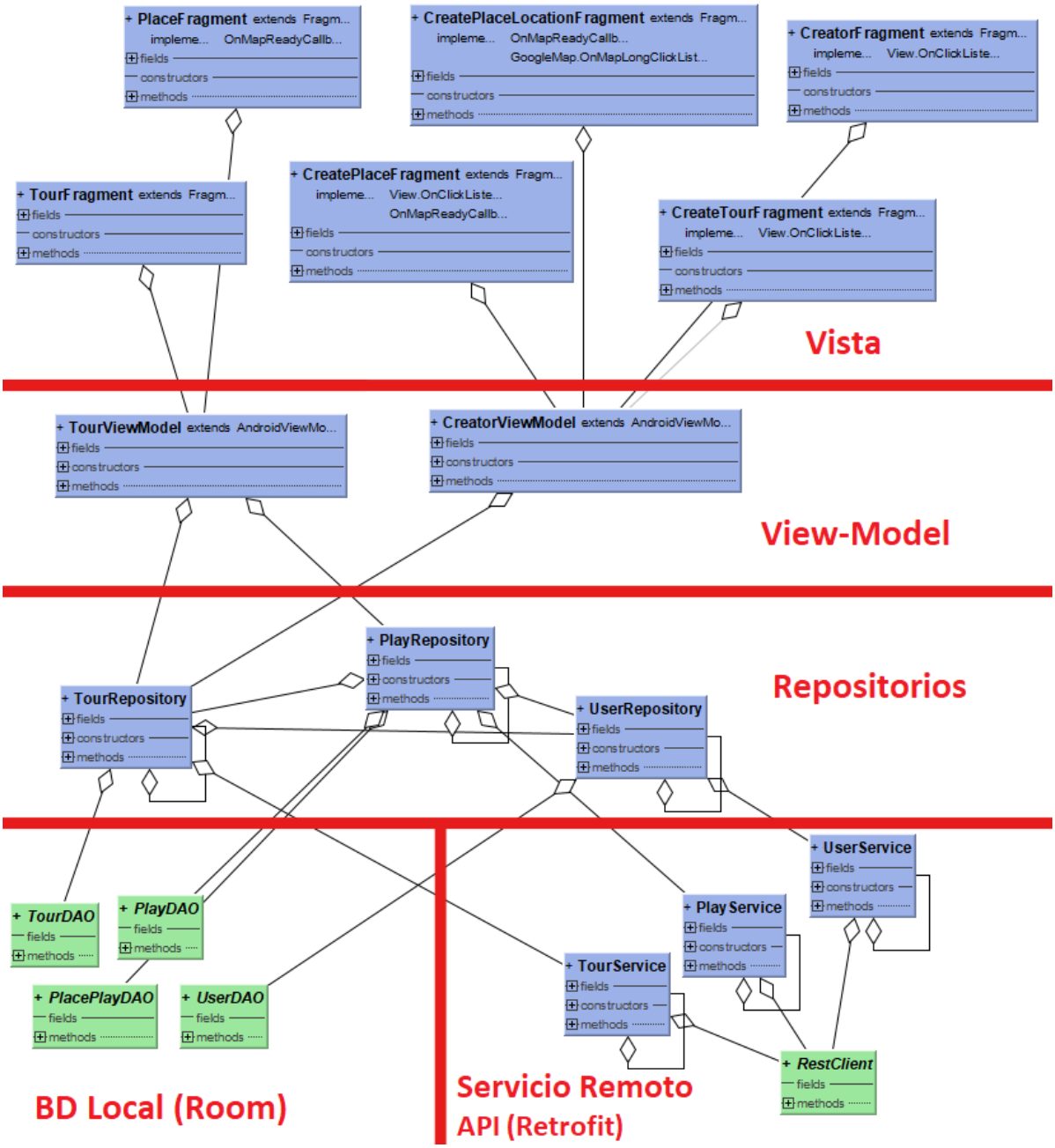

Figura 5.3.: Parte del diagrama de clases de la aplicación Android.

6. Implementación

6.1. Servidor

El framework Lumen define un fichero ("web.php") en el que se deben de incluir todas las rutas que atendemos.

Las rutas se pueden agrupar para indicar el prefijo que usan o para indicar que tienen que usar un middleware (ver sección 6.1.2) en el listado 6.1 lineas 1 y 2, se puede ver un ejemplo de uso.

```
1 $router->group(['prefix' => 'api'], function () use ($router) {
2     $router->group(['middleware' => 'jwt.auth'], function () use (
          $router) {
3         $router->put('users/{user_id}', ['uses' => '
              UserController@update']);
4         $router->get('users/{user_id}/plays', ['uses' => '
              UserController@getUserPlays']);
5     }
6 }
```

Listado 6.1: Ejemplo de la declaración de las peticiones.

En cada ruta se pueden usar parámetros opcionales, los cuales se representan en la propia ruta como `{user_id}`. Estos parámetros serán recibidos por el método que se indica. Por ejemplo, en la linea 4 del listado 6.1 tenemos una petición GET a la ruta `/api/users/{user_id}/plays`, que, haciendo uso del middleware jwt.auth, llamará al método `getUserPlays` del controlador `UserController`.

6.1.1. JWT (JSON Web Token)

Para realizar el proceso de atentificación de las peticiones a la API he utilizado la librería JWT de Firebase. Esta librería nos permite codificar y descodificar el token con una simple llamada a un método (ver listado 6.2).

Para codificar una petición le tendremos que pasar como primer parámetro los datos que queremos codificar en formato JSON y como segundo parámetro la contraseña a utilizar. De forma opcional se le puede pasar un tercer parámetro que será el algoritmo de cifrado a utilizar (por defecto es HS256 que utiliza HMAC con SHA-256[1]).

Para decodificar el método recibe otros tres parámetros. El primero de ellos es el token codificado, el segundo es la contraseña a utilizar y el tercero de ellos es un array con los algoritmos de cifrado a usar (lo intentará con todos hasta encontrar el que funciona). A continuación se incluye un ejemplo de como realizar este proceso de codificación y decodificación.

```
1 // Codificado
2 $token = JWT::encode($payload, $password, 'HS256');
3 // Decodificado
4 $credentials = JWT::decode($token, $password, ['HS256']);
```

Listado 6.2: Ejemplo de codificación y decodificación de JWT con la librería de Firebase.

6.1.2. Middleware

Un middleware es una capa de la aplicación por la que deberán pasar las peticiones que lo usen. Esto nos permitirá realizar acciones antes de que se ejecute la petición y también nos permitirá filtrar las peticiones a realizar. Por ejemplo, para verificar que el usuario ha iniciado sesión filtraremos las peticiones utilizando una de estas capas.

En el middleware de autenticación comprobamos la cabecera "Authorization" de la petición HTTP y de esta forma obtenemos el token JWT. Decodificando ese token obtendremos el id del usuario que ha iniciado sesión y la fecha en la que caducará este token. Si el token ha caducado o ese id no existe para ningún usuario, devolveremos el

[1]HMAC en la Wikipedia. `https://en.wikipedia.org/wiki/HMAC`

error correspondiente (ver sección 6.1.7).

Si todo ha ido bien, generaremos un nuevo token para ampliar la caducidad del token.

6.1.3. Google SSO

Para iniciar sesión con la cuenta de Google sin necesidad de contraseña, usaremos la librería Google Client[2] para PHP desarrollada por Google.

Para poder utilizar Google SSO deberemos dar de alta nuestro servicio en Google Cloud Platform[3], indicando qué información queremos obtener del usuario y crear unas credenciales de cliente OAuth 2.0. Esto generará un id de cliente que nos identifica para poder utilizar los servicios de Google.

Para usar estos servicios debemos crear un objeto de tipo `Google_Client` pasándole como parámetro un mapa con el id de cliente que obtuvimos de Google Cloud Platform (ver listado 6.3). Haciendo una llamada al método `verifyIdToken` pasándole como parámetro el token que se obtuvo en la aplicación, nos devolverá un mapa con los datos que tiene el token. Si ese token hubiese sido modificado para intentar suplantar la identidad de alguien, saltaría una excepción indicando que el token no es válido.

```
1 $client = new Google_Client(['client_id' => $client_id]);
2 $payload = $client->verifyIdToken($token);
3 $payload['sub'];
4 $payload['email'];
5 $payload['name'];
6 $payload['picture'];
```

Listado 6.3: Ejemplo de uso de la librería de Google para verificación de SSO.

[2]Google Client PHP en GitHub. `https://github.com/googleapis/google-api-php-client`
[3]Web de Google Cloud Platform. `https://console.cloud.google.com/apis/credentials`

6.1.4. Migraciones

Lumen nos provee de un sistema de migraciones[4] con las cuales crearemos las tablas de la base de datos así como sus relaciones. Cada migración hará una acción sobre la base de datos. Por ejemplo, si empezamos con una tabla con 4 columnas y necesitamos añadir una más, deberemos de crear una migración que la añada en vez de modificar la existente.

Como podemos ver en el siguiente listado (ver listado 6.4), la acción de crear la migración llamará al método `up` y al borrarla llamará a `down`. En este ejemplo se creará la tabla "tours" (ver imagen 6.1) con las siquientes columnas:

- Clave primaria 'id' autoincremental.
- 'name' de tipo varchar que no puede ser nulo.
- 'image' de tipo varchar que puede ser nulo.
- 'creator_id' de tipo entero grande sin signo (el mismo que una clave primaria) que puede ser nulo, es clave ajena y hace referencia a la columna id de la tabla "users" indicando que se debe borrar en cascada.
- 'min_level' de tipo enum con tres elementos y con un valor por defecto.
- El método timestamps creará dos columnas que se utilizarán solo internamente:
  - 'created_at' que nos indicará cuándo ha sido creado el registro de la base de datos.
  - 'updated_at' que hará lo mismo pero con la fecha de actualización.

```
1 class CreateToursTable extends Migration
2 {
3     public function up(){
4         Schema::create('tours', function (Blueprint $table) {
5             $table->bigIncrements('id');
6             $table->string('name');
7             $table->string('image')->nullable();
```

[4]Migraciones en Laravel. `https://laravel.com/docs/5.7/migrations`

```
8               $table->unsignedBigInteger('creator_id');
9               $table->foreign('creator_id')->references('id')->on('users'
                    )->onDelete('cascade');
10              $table->enum('min_level', ['therm', 'compass', 'map'])->
                    default('map');
11              $table->timestamps();
12          });
13      }
14      public function down(){
15          Schema::dropIfExists('tours');
16      }
17  }
```

Listado 6.4: Ejemplo de una migración.

El resultado de la migración anterior será el siguiente:

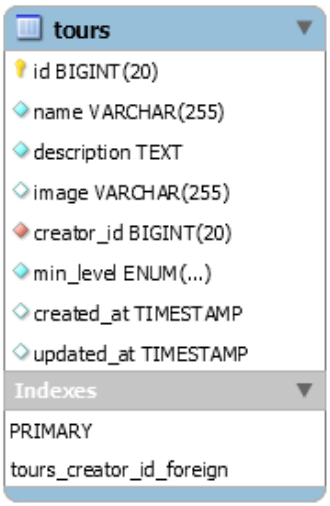

Figura 6.1.: Resultado de la migración 6.4.

El resultado de todas las migraciones nos generará el esquema de la imagen 5.1

6.1.5. Modelos

Para trabajar con entidades de base de datos como si fueran objetos, debemos de crear los modelos indicando como se relacionan entre ellos. Para ello deberemos de crear un

modelo[5] por cada una de las entidades, en mi caso se crearon los siguientes modelos: User, Tour, Place, Play y Social.

Por defecto Lumen tratará de buscar una tabla con el mismo nombre que la clase pero en minúscula y en plural (en inglés). Por ejemplo, para la clase User buscará la tabla users.

Dentro de la clase crearemos los métodos que nos darán acceso a las entidades relacionadas (ver listado 6.5). Por ejemplo, para una relación 1-N en la que un tour tiene un creador, definiremos la relación utilizando una función tipo: `belongsTo('el otro modelo')`. Para la inversa, N-1, se debe indicar con `hasMany`. Para definir una relación N-N tendremos que crear una tabla intermedia, y para definir esta relación utilizaremos el método `belongsToMany`, donde le indicaremos el otro modelo que relacionamos, la tabla intermedia y las dos columnas que forman la clave primaria, primero la de la tabla actual y después la de la otra tabla. Si no indicamos la tabla intermedia, Lumen entenderá que la tabla intermedia es una concatenación de las dos tablas que estamos relacionando, en orden alfabético y separadas por un guión bajo. Haciendo uso del método `withTimestamps` le indicamos que esa tabla intermedia tiene las marcas de tiempo de creación y actualización para que las cree y actualice.

```
1  class Tour extends Model {
2      public function creator() {
3          return $this->belongsTo('App\User');
4      }
5      public function places() {
6          return $this->hasMany('App\Place')->orderBy('order');
7      }
8      public function players() {
9          return $this->belongsToMany('App\User', 'plays', 'tour_id', '
               user_id')->withTimestamps();
10     }
11 }
```

Listado 6.5: Ejemplo de relación entre modelos.

[5]Modelos en Laravel. `https://laravel.com/docs/5.7/eloquent#defining-models`

6.1.6. Documentación de la API

Para utilizar la API desde el cliente hace falta tener una documentación donde se establezcan qué tipo de peticiones recibe, qué parámetros recibe, cuáles devuelve y para qué sirven.

Para ello he hecho uso de RAML (RESTful API Modeling Language), un modelo de datos con el formato YAML que permite establecer la documentación de una API REST indicando con todo detalle el formato que debe tener una petición y el tipo de respuesta que devuelve.

A continuación podemos ver un ejemplo de la documentación de un método de la API implementada, en la que se puede realizar una petición HTTP de tipo `GET` a la url `/tours`, es obligatoria la cabecera Authorization y obtenemos como resultado un listado de objetos con título y descripción.

```
/tours:
  get:
    description: Obtenemos todos los tours
    headers:
        Authorization:
            displayName: Authorization
            description: Token de verificación obtenido al iniciar sesi
                ón
            type: string
            required: true
    responses:
      200:
        description: Listado de tours
        body:
            application/json:
                example: |
                [
                    {
                        "titulo": "Paseo por Alicante",
                        "descripcion": "Paseo por el centro de Alicante
                            ..."
                    },
                    {
                        "titulo": "Paseo por Valencia",
                        "descripcion": "Paseo por el centro de Valencia
                            ..."
                    }
                ]
```

Listado 6.6: Ejemplo de documentación con RAML.

Para ver la documentación completa ver anexo B.

6.1.7. Errores

La gestión de los errores de las llamadas a la API se ha realizado utilizando el código de estado de la petición y añadiendo además un mensaje de error en el cuerpo de la respuesta. Si ocurriera un error en el servidor se devolvería el estado HTTP[6] correspondiente y el cuerpo en formato JSON con un atributo `type`, de tipo string, indicando el tipo de error, además de un campo `message` de tipo string con el mensaje de error.

6.2. Cliente

A continuación veremos cómo se ha implementado el cliente en Android. Al no hacer uso de un framework ha sido posible estructurar el código de la aplicación en base a nuestras necesidades. Además, en el anexo A se puede consultar el resultado final obtenido.

6.2.1. Interfaz

Una buena interfaz puede ser el aspecto decisivo en la elección entre dos aplicaciones que son iguales, ya que es lo que el usuario ve y con lo que interactúa a través de la pantalla de su dispositivo. Para conseguir un buen diseño pero sin necesidad de gastar horas en la creación de cada uno de los componentes, he decidido utilizar las guías de diseño creadas por Google llamadas Material Design [7].

Estas guías están orientadas a dar una buena usabilidad y diseño moderno. Haciendo

[6]Estados HTTP. `https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`
[7]Web Material Desgin. `https://material.io/design/`

uso de la librería para Android de Material Design[8], donde ya están creados todos los componentes necesarios para el desarrollo de una aplicación, me ha sido sencillo añadirlos a la interfaz y personalizarlos para el uso en mi aplicación.

Para la creación de las interfaces, Android Studio (ver sección 4.1.1.2) incluye un editor gráfico donde podremos añadir componentes arrastrándolos donde nosotros deseamos (ver imágenes 6.2 y 6.3), aunque también puede ser modificado el código XML que se genera. Posteriormente esta interfaz será cargada desde un fragmento (Fragment), y vinculado con las propiedades de este usando Butterknife (ver sección 4.1.1.6), donde se incluirá toda la funcionalidad.

Siguiendo las recomendaciones del equipo de desarrollo de Android, he hecho uso de la estrategia de usar una única actividad (Activity) en el que se irán mostrando los fragmentos haciendo uso de la navegación entre pantallas (ver sección 6.2.2).

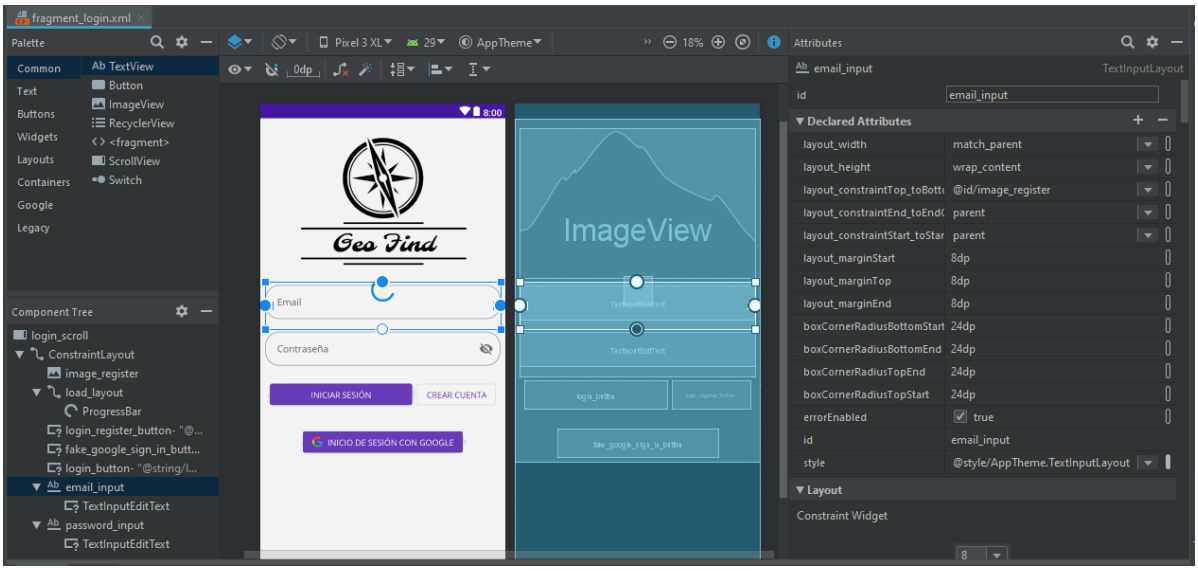

Figura 6.2.: Herramienta gráfica para el desarrollo de las interfaces en Android Studio.

[8]Material Design en GitHub. https://github.com/material-components/material-components-android

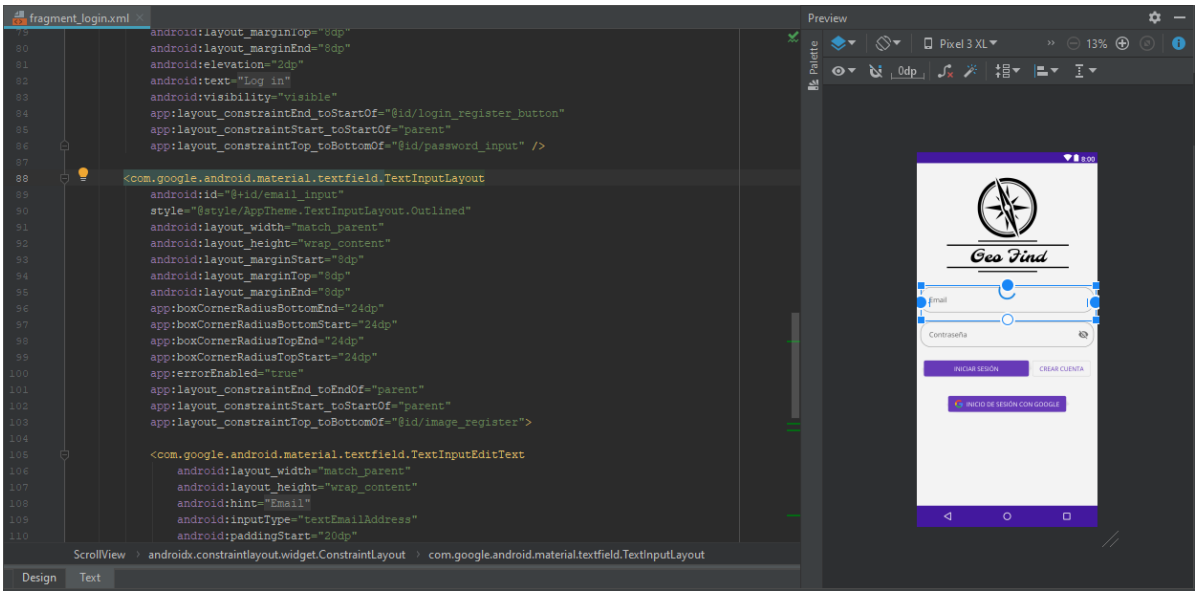

Figura 6.3.: Misma herramienta que en la imagen 6.2 pero en formato texto.

6.2.2. Navegación entre pantallas

Para aplicar la estrategia de usar una única actividad con múltiples fragmentos, se lanzó, en el conjunto de librerías de Android Jetpack (ver sección 4.1.1.3), la librería Navigation[9], la cual nos permite tener una fácil transición entre las diferentes vistas de nuestra aplicación con solo configurar un fichero XML.

Para facilitar la creación de este fichero donde configuramos las transiciones entre las diferentes vistas (fragmentos), se añadió en Android Studio 3.3 una herramienta gráfica (ver imagen 6.4) la cual hace el trabajo más fácil. Al igual que la herramienta para crear la interfaz (ver sección 6.2.1), esta también tiene el modo de texto.

Cada una de las vistas se debe de configurar con un id y cada una de las acciones que nos llevan a otras vistas deben de tener otro id. Cuando queremos realizar una de estas acciones deberemos hacerles referencia a través de su id. En el panel izquierdo podemos configurar los argumentos que recibe la vista y qué acciones se pueden hacer desde ella.

[9]Navegación en Android. `https://developer.android.com/guide/navigation/navigation-getting-started`

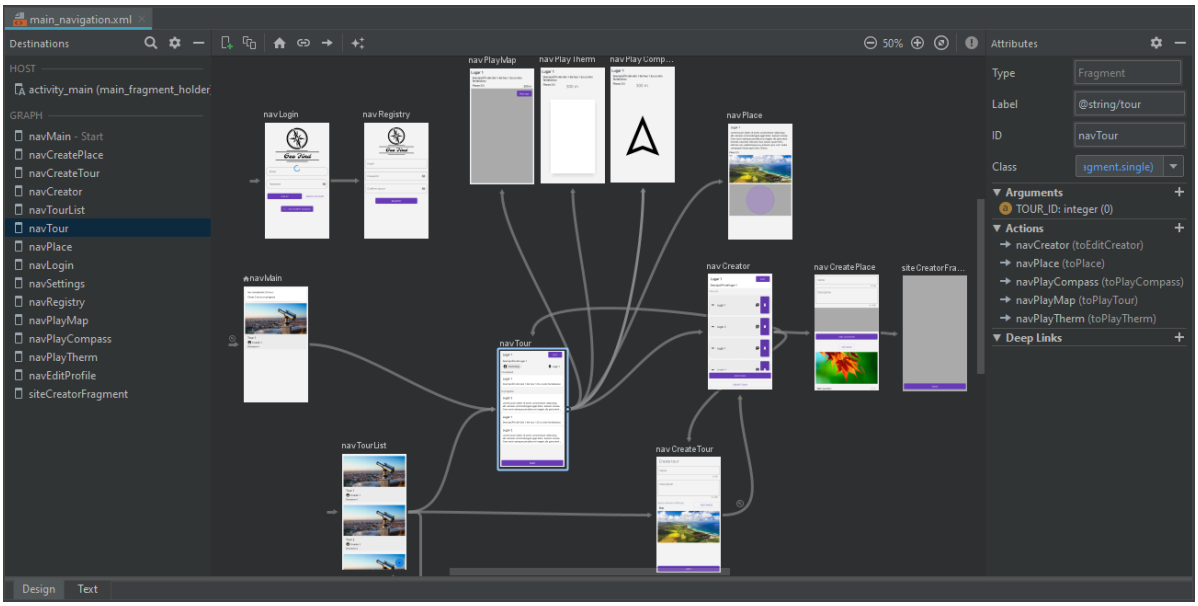

Figura 6.4.: Herramienta gráfica para modificar las transiciones entre vistas.

6.2.3. LiveData

A la hora de mostrar la interfaz se deben cargar los datos que el usuario necesita y, si ocupamos el hilo de ejecución con esta tarea la aplicación, se quedaría congelada hasta recibir los datos. Para ello se deben cargar los datos de forma asíncrona (ver imagen 6.5) y utilizar los objetos LiveData, en concreto los MutableLiveData.

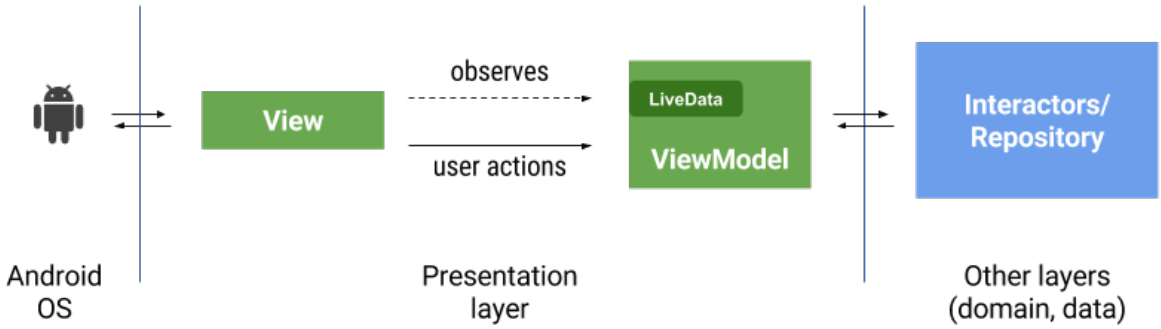

Figura 6.5.: Carga asíncrona de datos usando LiveData.

Como podemos ver el listado 6.7, se haría la llamada al método getTour del ViewModel indicando que cuando se reciban los datos se llamará al método setTour, el cual recibe

como parámetro el tour y establece los valores necesarios en la vista.

```
    viewModel.getTour(tour_id, user.getId())
             .observe(this, this::setTour);
```

Listado 6.7: Ejemplo de uso de objetos LiveData desde el fragmento.

En el ViewModel deberíamos crear el objeto MutableLiveData tipado al dato que debemos devolver y, de forma asíncrona, hacer la llamada al repositorio como se puede ver en el listado 6.8. Devolveríamos el objeto MutableLiveData y, cuando tenemos el tour cargado, es cuándo lo enviamos a través del objeto MutableLiveData que habíamos devuelto.

```
public MutableLiveData<Tour> getTour(int tour_id, int user_id) {
    MutableLiveData<Tour> m = new MutableLiveData<>();
    new Thread(() -> {
        try {
            tour = tourRepo.getTour(tour_id);
        } catch(APIException e) {
            setError(e.getType());
            tour = null;
        }
        m.postValue(tour);
    }).start();
    return m;
}
```

Listado 6.8: Ejemplo de uso de objetos LiveData desde el ViewModel.

6.2.4. Retrofit

Para poder mantener una comunicación con el servidor y abstraernos de la complejidad que conlleva la creación de las peticiones, he hecho uso de la librería Retrofit que nos reduce significativamente la cantidad de código a implementar.

La librería Retrofit se basa en una interfaz de Java en la cual definiremos tantos métodos como peticiones queramos hacer. A continuación se pueden ver algunos ejemplos de métodos que habría en esta interfaz:

```
@POST("tours")
```

```
2  Call<Tour> createTour(@Body Tour tour);
3  @GET("tours/{tour_id}")
4  Call<Tour> getTour(@Path("tour_id") Integer tour_id);
5  @PUT("tours/{tour_id}")
6  Call<Tour> update(@Path("tour_id") Integer tour_id, @Body Tour tour);
```

Listado 6.9: Ejemplo de métodos de la interfaz de Retrofit.

Haciendo uso de las etiquetas o decoradores indicaríamos qué método HTTP debe de generar y sobre qué ruta. Dado que las peticiones deben de ser asíncronas, el objeto que devuelven es de tipo Call y tipado al tipo de objeto que queremos. A través de los parámetros que recibe el método le indicaremos los remplazos en la URL que debe hacer con la etiqueta `@Path`. Si la petición tiene cuerpo entonces el objeto que formará el cuerpo será etiquetado como `@Body`. En caso de ser necesarios los parametros de la query, entonces se indicará con la etiqueta `@QueryMap` y el objeto pasado será de tipo `Map<String, Sting>`.

Para hacer uso de los métodos es necesario crear el objeto que implementa la interfaz a través del constructor de Retrofit `new Retrofit.Builder()`[10]. A continuación podemos ver el uso de este objeto para realizar la llamada definida en la figura 6.9.

```
1  public Tour getTour(Integer id) throws APIException {
2      Response<Tour> response;
3      APIException apiException;
4      try {
5          response = restClient.getTour(id).execute();
6          if(response.isSuccessful()) {
7              return response.body();
8          }
9          apiException = ErrorUtils.parseError(response);
10     } catch(Exception e) {
11         apiException = new APIException(ErrorType.NETWORK, e.getMessage
               ());
12         Log.e(TAG, "getTour: ", e);
13     }
14     throw apiException;
15 }
```

Listado 6.10: Ejemplo de llamada a la API a través de Retrofit.

Para manejar el envío de la cabecera "Authentication" he añadido un interceptor a to-

[10]Documentación de Retrofit. `https://square.github.io/retrofit/`

das las peticiones salientes añadiéndole esta cabecera con el token que obtiene de las preferencias del usuario (`getBearerToken()`). Como el servidor renueva el token en cada petición, este interceptor también se encarga de almacenar este nuevo token (`Preferences.setToken(sp, newToken)`) y en caso de que falle porque ha caducado ejecutaría de nuevo la acción de login con las credenciales del usuario almacenadas en las preferencias, obteniendo un token nuevo y ejecutando de nuevo la petición inicial que falló (Ver listado 6.11).

```
1  OkHttpClient okHttpClient = new OkHttpClient.Builder().addInterceptor(
       chain -> {
2      Request originalRequest = chain.request();
3      Request newRequest = originalRequest.newBuilder()
4              .addHeader("Authorization", getBearerToken())
5              .method(originalRequest.method(), originalRequest.body()).
                  build();
6
7      Response response = chain.proceed(newRequest);
8      if(response.code() == 401) { // Code 401 Unauthorized
9          try {
10             if(userRepo == null) {
11                 userRepo = UserRepository.getInstance();
12             }
13             userRepo.reLogin();
14         } catch(APIException e) {
15             Log.e(TAG, "Fail relogin", e);
16         }
17         newRequest = originalRequest.newBuilder()
18                 .addHeader("Authorization", getBearerToken())
19                 .method(originalRequest.method(), originalRequest.body
                       ()).build();
20         response = chain.proceed(newRequest);
21     } else {
22         String newToken = response.header("Authorization");
23         if(newToken != null && !newToken.isEmpty()) {
24             Preferences.setToken(sp, newToken);
25         }
26     }
27     return response;
28 }).build();
```

Listado 6.11: Algoritmo de renovación de token en todas las peticiones y login en caso de fallo.

6.2.5. Google SSO

Para facilitar el inicio de sesión de los usuarios, Google ofrece el servicio de SSO (Single-Sign-On) a través del cual las aplicaciones obtienen un identificador único para cada usuario en cada aplicación. Haciendo uso de ese identificador podremos validar a nuestro usuario sabiendo que ha iniciado sesión con Google.

Google añade una librería con todo lo necesario para obtener los datos del usuario en la aplicación[11] y nosotros somos los encargados de mandar esta información a nuestro servidor. Al añadir el botón de inicio de sesión de Google a nuestra interfaz (ver imagen 4.12) tendremos acceso al objeto `GoogleSignInAccount`, de donde podremos obtener toda la información que hayamos indicado que necesitamos al registrar nuestra aplicación (ver sección 6.1.3). En este caso obtendremos el email, el identificador del usuario y el token. Mandaremos el email y el token al servidor, y con eso podremos iniciar sesión o registrarlo si no lo está.

6.2.6. Integración continua

Para poder hacer uso de la integración continua con Travis, primero deberemos de crearnos una cuenta en su web (ver sección 3.5). A continuación indicaremos que repositorio queremos que analice. Para configurar la ejecución, tendremos que crear un fichero .travis.yml como el del listado 6.12.

Con la sección `brances` indicamos que solo queremos ejecutar la integración en las ramas ahí indicadas. A continuación tenemos la sección `android` donde indicaremos los componentes que necesitaremos, en nuestro caso será la distribución android SDK 29, sus build-tools, la librería extra para los servicios de Google Play y los repositorios para descargar las librerías que usamos. La parte principal es la sección `script`, en la cual indicaremos los comandos que debe de ejecutar, en nuestro caso se construirá el proyecto con `gradlew build` y se generará el informe sobre cobertura de código con `gradlew jacocoTestReport` gracias al plugin de grale JaCoCo[12] (Java Code Coverage). Si queremos ejecutar algo después del script tenemos la sección `after_success`, que se ejecutará solo si

[11] Documentación del SSO de Google en Android. `https://developers.google.com/identity/sign-in/android/`

[12] Librería JaCoCo. `https://www.jacoco.org/jacoco/`

la ejecución ha sido satisfactoria (ha compilado correctamente, ver sección 3.5). En este caso vamos a ejecutar el código que nos provee la web CodeCov para después publicar los resultados de cobertura de código en GitHub (ver sección 3.5).

```
language: android  # Lenguaje del código
dist: trusty       # Distribución de Linux en la que ejecutaremos

branches:
  only:
    - master
    - dev

android:
  components:
    - build-tools-29.0.2
    - android-29
    - extra-google-google_play_services
    - extra-google-m2repository
    - extra-android-m2repository

script:
  - ./gradlew clean build
  - ./gradlew clean jacocoTestReport

after_success:
  - bash <(curl -s https://codecov.io/bash)
```

Listado 6.12: Ejemplo de fichero .travis.yml.

6.2.7. Geolocalización por GPS y orientación magnética

Para implementar el seguimiento de un tour mediante la brújula, he utilizado la librería `SensorManager`[13]. Mediante esta librería tendremos acceso a las actualizaciones con los datos del sensor magnético.

Estos datos del sensor vienen dados en μT (micro Tesla) para los tres ejes de coordenadas (X, Y y Z). Por lo que tendremos que transformarlos a un solo ángulo de giro para mostrarlo en la brújula. Para simplificar este trabajo la librería SensorManager

[13]SensorManager en la documentación de Android. `https://developer.android.com/reference/android/hardware/SensorManager`

tiene un par de métodos (`getRotationMatrix`[14] y `getOrientation`[15]), que con el dato del sensor magnético y el del acelerómetro, nos devuelve un array con el ángulo de giro en los tres ejes con respecto al Norte, de los cuales solo nos hace falta el del eje z (ángulo azul en la figura 6.6).

También necesitamos la orientación hacia el destino, para ello usamos los objetos Location[16]. Teniendo la localización del usuario debemos de llamar al método `bearingTo()`, pasándole la localización del destino, que nos devolverá el ángulo que forma la trayectoria desde donde está el usuario hacia el destino con referencia al Norte (ángulo amarillo en la imagen 6.6).

En la siguiente imagen podemos ver el funcionamiento del algoritmo. Nosotros estamos ubicados en la flecha azul, la linea roja marca la dirección del norte, la línea azul marca nuestro sentido de orientación y la línea amarilla la dirección al destino. Sumando esos dos ángulos obtendremos el ángulo para mostrar en la brújula.

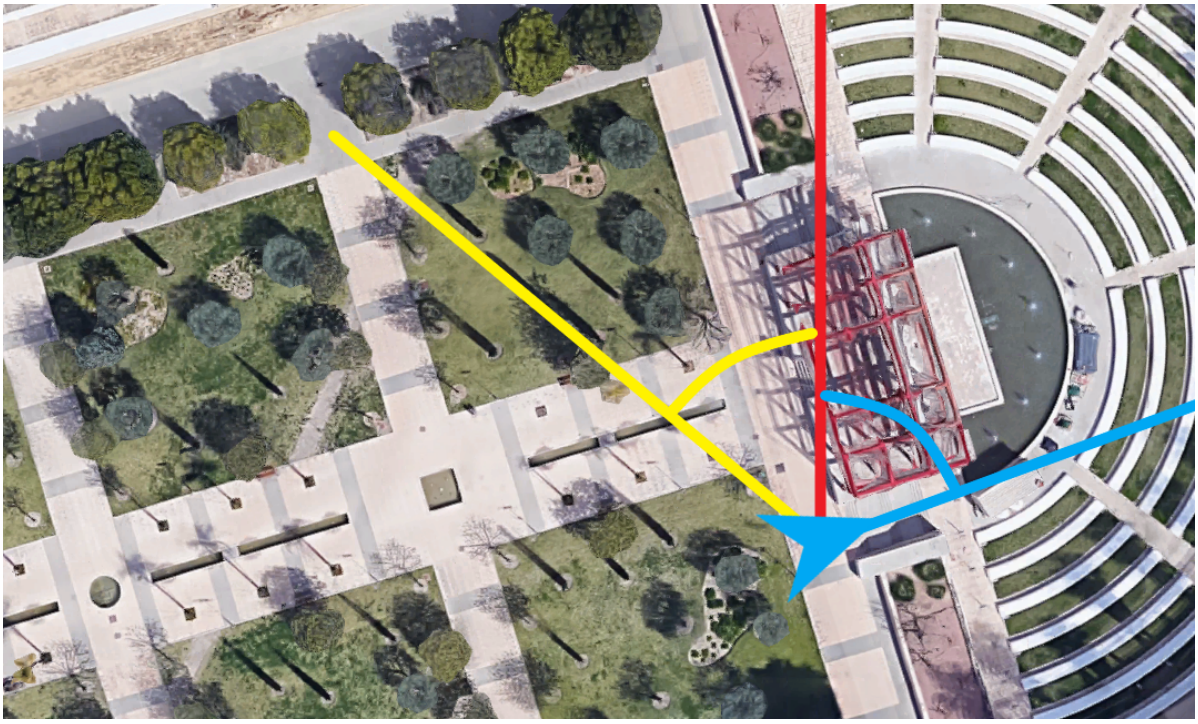

Figura 6.6.: Ejemplo de cómo se relacionan los ángulos.

[14]Definición en la documentación de Android. getRotationMatrix()
[15]Definición en la documentación de Android. getOrientation()
[16]Location en la documentación de Android. `https://developer.android.com/reference/android/location/Location`

7. Conclusiones

Concluyendo con este proyecto voy a repasar el trabajo realizado durante el mismo.

Ha sido un gran reto desarrollar una aplicación para el sistema operativo Android sin una experiencia previa pero ha merecido la pena ya que he aprendido solo y me he dado cuenta de mis propios errores.

Retomando los objetivos indicados en la sección 1.1, los objetivos cumplidos son los siguientes:

- Desarrollar un código limpio, modular y reutilizable: se ha llevado a cabo una estructura de código mediante arquitecturas conocidas y muy utilizadas. Además, se han realizado análisis y refactorizaciones constantes para mantener un código limpio.

- Establecer integración continua tanto en cliente como en servidor: como hemos visto en la sección 6.2.6, se ha llevado a cabo la integración continua en cliente y servidor.

- Realizar más de release: hemos realizado seis entregas de forma periódica (ver sección 3.4).

- Mantener el código documentado con comentarios: se han realizado los comentarios necesarios en los repositorios y servicios del cliente además de en todos los métodos de los controladores del servidor.

- Hacer uso de las últimas versiones de librerías y frameworks: A día 1 de septiembre, se está utilizando la última versión estable de todas las librerías en uso.

- Mostrar un listado de tours guiados disponibles: Se está mostrando un listado con todos los tours que están disponibles para el usuario.

- Búsqueda de tours: Dentro del mismo listado de tours se permite al usuario realizar un filtrado por el texto del título o descripción del tour (ver anexo A).

- Realización de tours mostrando indicaciones: El usuario puede realizar un tour de hasta tres maneras diferentes (ver anexo A).

- Guardar estado de los tours realizados: Se almacena el progreso del tour para cada usuario.

- Desarrollo de las pantallas para el uso, creación y modificación de los tours: Desarrolladas las pantallas para la vista, creación, modificación y seguimiento de tours (ver anexo A).

Por lo tanto se han cumplido todos los objetivos propuestos inicialmente. Sin embargo, sí que quedaría mucho trabajo por delante para conseguir una aplicación de calidad. Como trabajo futuro esta aplicación necesita:

- Cobertura de código de al menos un 80 % de código con los tests.

- Una mejor experiencia de usuario, teniendo en cuenta los diferentes tamaños de pantalla que puede tener un dispositivo.

- Añadir más seguridad en la comunicación entre el cliente y el servidor y a la hora de almacenar los datos en la base de datos.

Además también resulta relevante comentar que el código fuente ha sido publicado en mi cuenta de GitHub, el cual puede servir de ayuda a la comunidad de usuarios. Cliente: `https://github.com/martinlaizg/geo-find-client`

Servidor: `https://github.com/martinlaizg/geo-find-server`

Bibliografía

[1] Android Developers. Documentación propia de android. https://developer.android.com/about/dashboards.

[2] Marcus Pöhls. Getting started and creating an android client. https://www.futurestud.io/tutorials/retrofit-getting-started-and-android-client.

[3] William J. Francis. Create your own magnetic compass using android's internal sensors. https://www.techrepublic.com/article/pro-tip-create-your-own-magnetic-compass-using-androids-internal-sensors/.

[4] Laravel. Laravel documentation. https://laravel.com/docs/5.7.

[5] Lumen. Lumen documentation. https://lumen.laravel.com/docs/5.7.

[6] Antonio Javier Gallego. Laravel 5. https://ajgallego.gitbooks.io/laravel-5/content/introduccion.html.

A. Imágenes de la aplicación final

A continuación se mostrarán algunas imágenes de la aplicación en funcionamiento.

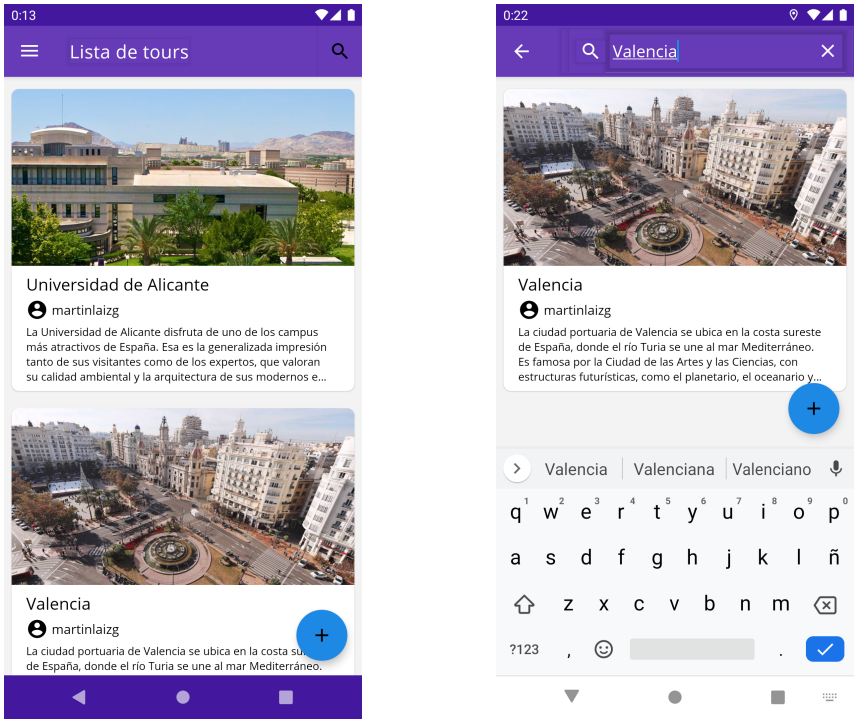

Figura A.1.: Listado y buscador de tours.

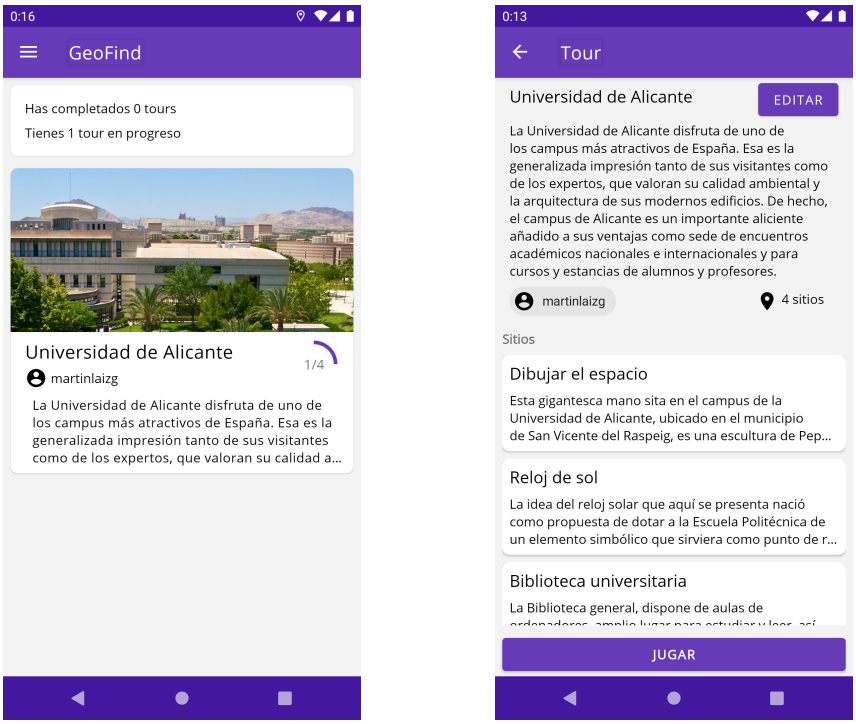

Figura A.2.: Tours en progreso y vista de un tour.

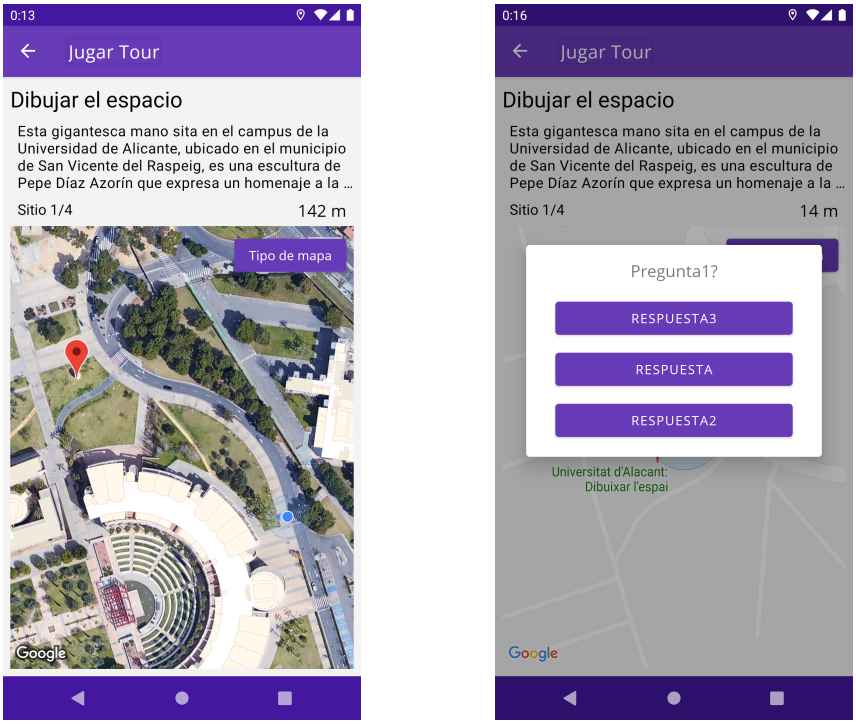

Figura A.3.: Seguimiento de tour con mapa y pregunta al completar.

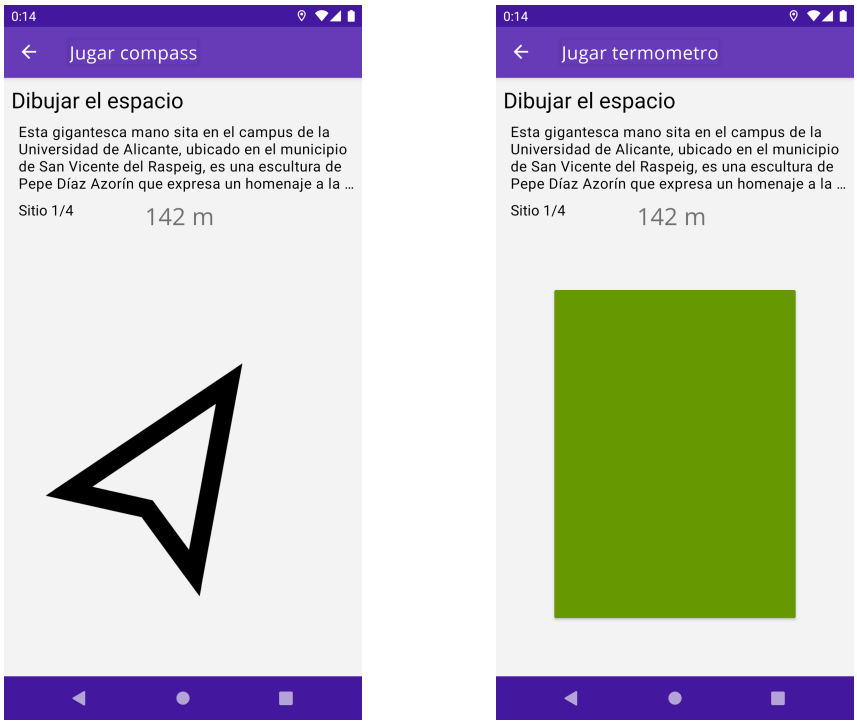

Figura A.4.: Guiado de tour con brújula y termómetro.

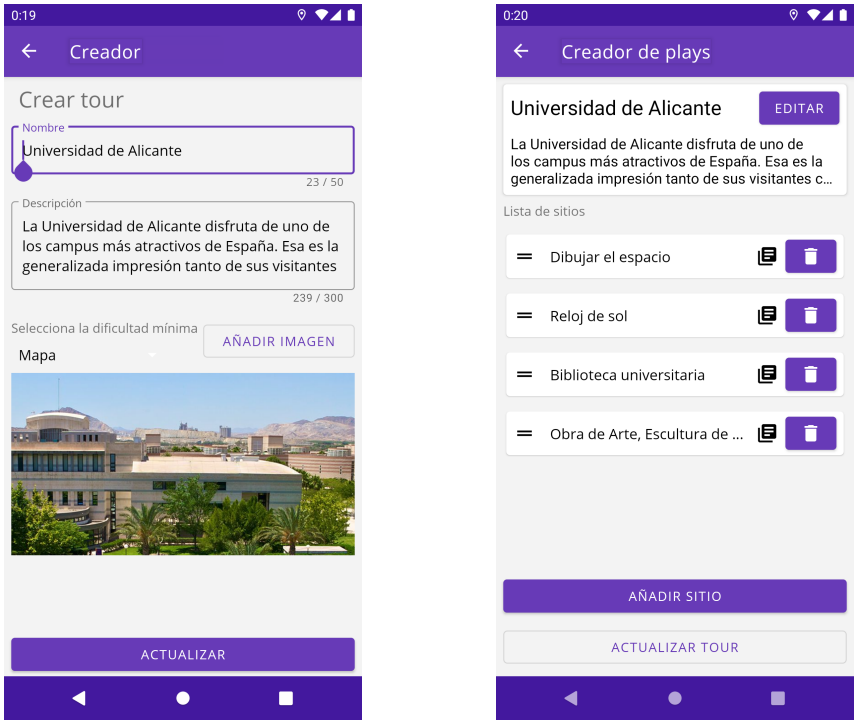

Figura A.5.: Creador de tour.

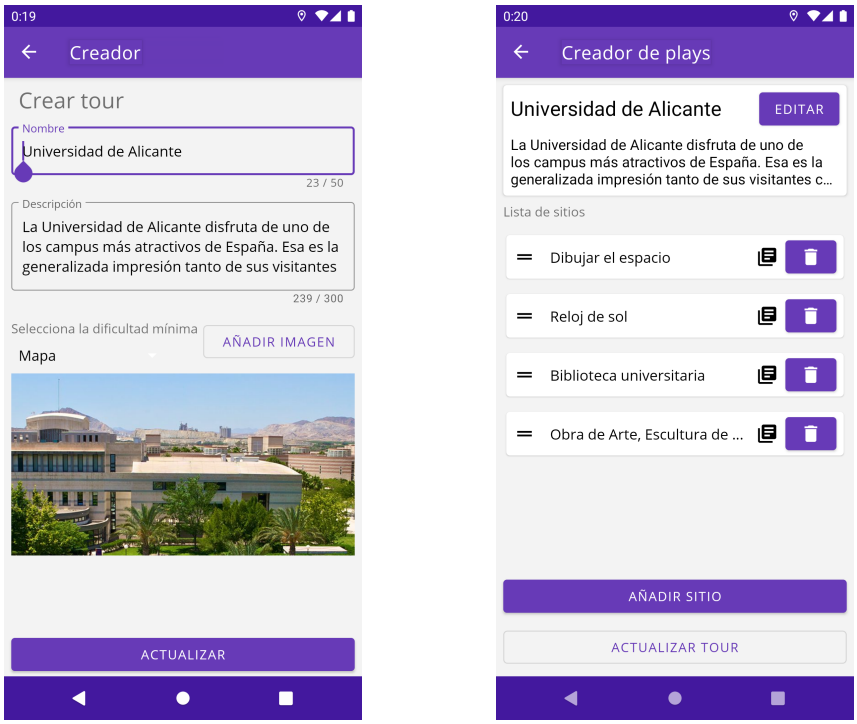

Figura A.6.: Creador de los sitios del tour.

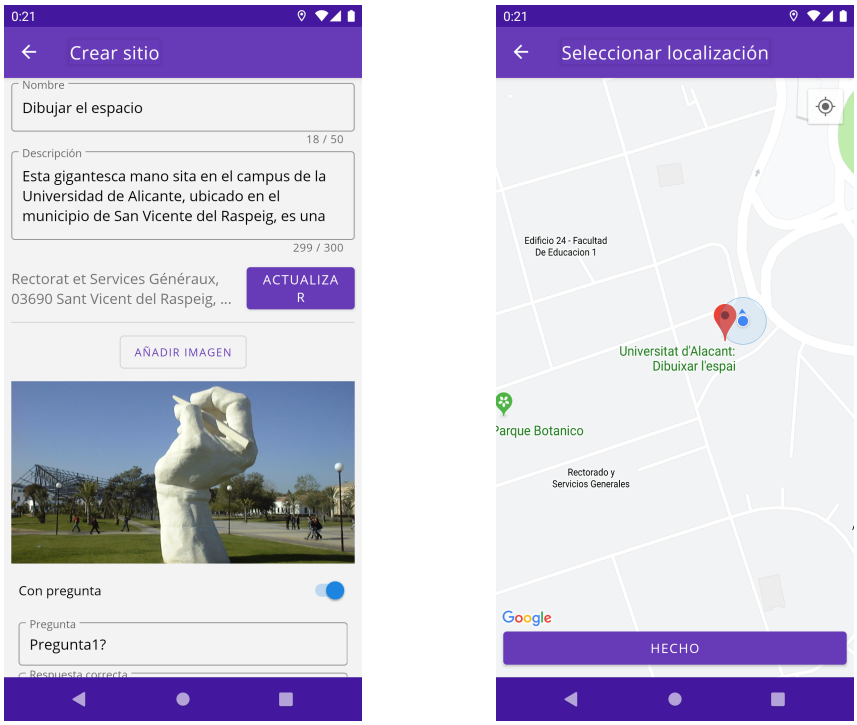

Figura A.7.: Modos de seguimiento de un tour.

B. Documentación de la API

A continuación se muestra la documentación de la API en lenguaje RAML.

```
#%RAML 1.0
title: GeoFind API
version: v1
baseUri: https://geofind1.herokuapp.com/api
mediaType:  application/json

types:
  Place:
    type: object
    properties:
      id: integer
      tour_id: integer
      tour?: Tour
      name: string
      description: string
      question: string
      answer: string
      answer2: string
      answer3: string
      order: integer
      lat: number
      lon: number
      image: string
      created_at: datetime
      updated_at: datetime
  Tour:
    type: object
    properties:
      id: integer
      name: string
      description: string
      image: string
      creator_id: integer
      min_level: string
      creator?: User
      places?: Place[]
```

```
        created_at: datetime
        updated_at: datetime
    Play:
      type: object
      properties:
        id: integer
        tour_id: integer
        user_id: integer
        created_at: datetime
        updated_at: datetime
        user: User
        places: Place[]
        tour: Tour
    User:
      type: object
      properties:
        id: integer
        email: string
        username: string
        name: string
        image: string
        user_type: string
        created_at: datetime
        updated_at: datetime
    ErrorResponse:
      type: object
      properties:
        type: string
        message: string
    Login:
      type: object
      description: Objeto login con email y
        si el provider es "own" (propio) el secure será la contraseña
           hasheada
        si el provider es "google" el secure será el token del usuario.
      properties:
        email: string
        secure: string
        provider: string


traits:
  withError:
    responses:
      400:
        description: Error generico, ver el campo type para comprender
           de que se trata
        body:
          type: ErrorResponse
  logged:
    responses:
```

```
86        200:
87          headers:
88            Authorization:
89              description: Nuevo token de autorización
90              type: string
91        401:
92          description: No hay token, el token ha expirado o es incorrecto
93          body:
94            type: ErrorResponse
95        404:
96          description: El usuario del token no existe
97          body:
98            type: ErrorResponse
99      headers:
100       Authorization:
101         displayName: Authorization
102         description: Token de verificación obtenido al iniciar sesión
103         type: string
104         required: true
105         example: "Bearer eyJ0eXAiOiJKV1QiLCJhbGci..."
106
107 /tours:
108   get:
109     description: Obtener todos los tours
110     is: [withError]
111     responses:
112       200:
113         description: Listado de tours
114         body:
115           application/json:
116             type: Tour[]
117   post:
118     description: Crea un tour
119     is: [withError,logged]
120     body:
121       application/json:
122         type: Tour
123     responses:
124       200:
125         description: Tour creado
126         body:
127           application/json:
128             type: Tour
129   /{id}:
130     get:
131       description: Obtener el Tour con el id correspondiente
132       is: [withError,logged]
133       responses:
134         200:
135           description: Tour con el id correspondiente
136           body:
```

```
137               application/json:
138                 type: Tour
139       put:
140         description: Actualiza los datos de un Tour
141         is: [withError,logged]
142         body:
143           application/json:
144             type: Tour
145         responses:
146           200:
147             description: Tour con los datos actualizados
148             body:
149               application/json:
150                 type: Tour
151       /places:
152         get:
153           description: Obtiene las localizaciones de un Tour
154           is: [withError]
155           responses:
156             200:
157               body:
158                 application/json:
159                   type: Place[]
160       /users:
161         /{user_id}:
162           /play:
163             get:
164               description: Obtiene el progreso del usuario con id user_id
                      en el tour con id tour_id
165               is: [withError,logged]
166               responses:
167                 200:
168                   body:
169                     application/json:
170                       type: Play
171                 403:
172                   description: No tienes permisos para acceder porque el
                        id del token no coincide con el user_id
173                   body:
174                     type: ErrorResponse
175             post:
176               description: Crea la relación de un usuario con un Tour
                     para empezar a seguirlo
177               is: [withError,logged]
178               responses:
179                 200:
180                   body:
181                     application/json:
182                       type: Play
183   /users:
184     post:
```

```
185       description: Crea un usuario
186       is: [withError]
187       body:
188         application/json:
189           type: User
190       responses:
191         200:
192           body:
193             application/json:
194               type: User
195     /{user_id}:
196       put:
197         description: Actualiza los datos de un usuario
198         is: [withError,logged]
199         body:
200           application/json:
201             type: User
202         responses:
203           200:
204             body:
205               application/json:
206                 type: User
207       /plays:
208         get:
209           description: Obtiene el progreso del usuario con id user_id
                para el tour con id tour_id
210           is: [withError,logged]
211           responses:
212             200:
213               body:
214                 application/json:
215                   type: Play
216   /plays:
217     /{play_id}:
218       get:
219         description: Obtiene el progreso de un usuario en un tour dado el
              id de la relación
220         is: [withError,logged]
221         responses:
222           200:
223             body:
224               application/json:
225                 type: Play
226       /places:
227         /{place_id}:
228           post:
229             description: Completa una localización con id place_id para
                  un tour en progreso dado el id play_id
230             is: [withError,logged]
231             responses:
232               200:
```

```
                body:
                  application/json:
                    type: Play
/login:
  post:
    description: Inicia sesión de un usuario
    is: [withError]
    body:
      application/json:
        type: Login
    responses:
      200:
        body:
          application/json:
            type: User
/support:
  post:
    description: Manda un mensaje con un informe de fallo o sugerencia
       del usuario
    is: [withError,logged]
    body:
      application/json:
        properties:
          title: string
          message: string
    responses:
      200:
```